Data Management

OR

GETTING PUNCHED IN THE FACE BY SQL AND PANDAS

BY

NICHOLAS ROSS, PHD

PROFESSOR OF DATA SCIENCE
UNIVERSITY OF CHICAGO

© 2020-2023 All Rights Reserved

Cover Art By: Megan Carlsen

Version: 2023-08-22 21:45:36

Contents

Introduction and Errata i

Relational Databases

1 Rows and Columns 1
- 1 What is a Relational Database . . . 3
- 2 Selecting Columns . . . 6
- 3 WHERE: Filtering rows . . . 7
- 4 Null . . . 9
- 5 ORDER BY and LIMIT . . . 11
- 6 Column Numbering . . . 16
- 7 Where are we: A Note on Scope . . . 17

2 Basic Manipulations 19
- 1 Types . . . 21
- 2 Renaming a Column . . . 23
- 3 Basic Mathematical Manipulations, ABS and LEAST/GREATEST . . . 24
- 4 Queries without a FROM Clause and Singletons . . . 28
- 5 String Functions: LEFT, RIGHT, LOWER, UPPER, LENGTH, TRIM and CONCAT . . . 29
- 6 ROUND and Changing Types (CAST) . . . 33
- 7 CAST and changing types . . . 33

3 Subqueries, Distinct & Case 41
- 1 Query Evaluation Order: SELECT and WHERE . . . 43
- 2 Comparisons: BETWEEN, LIKE and ILIKE . . . 45
- 3 CASE: Conditional Logic . . . 47
- 4 The DISTINCT Operator . . . 52
- 5 Subqueries (IN, ANY, ALL) . . . 55
- 6 Correlated Subqueries . . . 58

4 Database Internals: Transactions 63
- 1 REDO / COMBINE NEXT SECTIONS . . . 65
- 2 Table Creation and Deletion . . . 65
- 3 Database Operations: CRUD . . . 65
- 4 Creating Tables, Constraints and Deleting tables . . . 66
- 5 Altering Tables . . . 68
- 6 Inserting, Copying, Updating and Deleting . . . 68
- 7 Transactions and ACID . . . 69
- 8 Isolation Levels in Relational Databases . . . 73
- 9 Why do we care (NoSQL)? . . . 78

DRAFT

10 NoSQL . . . 80
11 Transaction Implementations [TBD] . . . 81

5 Aggregations **83**
1 Introduction to MTA data set . . . 85
2 GROUP BY clause . . . 86
3 Column numbering syntax . . . 91
4 Aggregates and CASE Statements . . . 93
5 Named Subqueries . . . 95

6 Dates and Types **101**
1 Date Types . . . 103
2 Date Functions . . . 104
3 Hard GROUP BY problems . . . 110

7 Averages **115**
1 The Trouble with Averages . . . 117
2 HAVING . . . 119
3 COALESCE and NVL . . . 120

8 Joins **123**
1 Joins . . . 125
2 UNION and UNION ALL . . . 132
3 Best Practices when Combining Tables . . . 134
4 Intermediate Joins . . . 136
4.1 Aggregations on-self . . . 136
4.2 Cross Joins for missing values . . . 137
5 Statistical Analysis in SQL . . . 138

9 Advanced Joins **147**
1 The Shape of Data . . . 149
2 Revenue over time & Advanced Joins . . . 151
2.1 First Value . . . 152
2.2 Most common value by group . . . 156
2.3 Cumulative Sum . . . 158
2.4 Rolling 90 day Calculation . . . 160
2.5 Cohorted Monthly Revenue . . . 161

10 Analytic Functions & CTE's **165**
1 Analytic Functions . . . 167
2 Using Analytic Functions with Transaction Data . . . 174
3 Common Table Expressions ("CTE") . . . 176
4 CTEs with the transaction data . . . 178

11 Database Internals: Performance Evaluation **181**
1 Normalization . . . 183
2 Views . . . 185
3 Information Schema . . . 189
4 Performance Considerations . . . 190
5 Index . . . 194
6 Distributed Systems and the CAP Theorem . . . 195

DRAFT

12 Extensions [TBD] 199
1 More Advanced Joins . . . 204
2 OLAP: Cube and Rollup . . . 210
3 Schemas . . . 210
4 Keys . . . 210
5 Data Exploration Strategies . . . 210
6 Query Strategies . . . 210

13 Interview Hints 211
1 Interview Hints . . . 212
2 Example Interview #1 . . . 216
3 Example Interview #2 . . . 218
4 Example Interview #3 . . . 220
5 Example Interview #4 . . . 221

Pandas

14 Introduction 223
1 What is Pandas . . . 225
2 Data structures . . . 226
3 Selecting Columns and Rows . . . 231
4 Column Types Conversion . . . 237
5 Dealing with NaN . . . 237
6 Choosing the largest and smallest values . . . 239
7 Manipulating Data & Method Chaining . . . 240
8 Indexes: Creating and Dropping . . . 244
9 Views and Copies . . . 246

15 More Manipulations and Types 251
1 Sorting DataFrames . . . 253
2 Dealing with Duplicates . . . 256
3 Using Type specific functions . . . 258
3.1 Dates . . . 258
3.2 Strings . . . 260
4 CASE style statements and the "isin" operator . . . 264
5 Regex Pattern Matching . . . 265

16 Aggregations 269
1 Introduction to the MTA dataset . . . 271
2 Simple Aggregations . . . 271
3 GroupBy Objects . . . 274
4 Advanced Index / Multiindex . . . 279
5 If not indexes... . . . 285
6 Indexing with aggregations, a big Gotcha . . . 286

17 Joins 289
1 Helpful Table / Review . . . 291
2 Merging data in Pandas . . . 292
3 Complex Join Conditions . . . 294
4 Stacking Data . . . 294

DRAFT

5 Lags and Leads . . . 296
6 Apply, map and applymap: Advanced Transformations . . . 297

18 Window Functions **301**
1 Window Functions in Pandas . . . 303
2 Some gotchas . . . 307
3 Reshaping Data: Transpose, Stack and Unstack . . . 308
4 A Bunch of stuff to clean up . . . 312
5 Combining with the original DataFrame . . . 312
6 Moving the Window . . . 316
7 Pivot / Melt . . . 316

Appendix

Appendix A Data Dictionaries **317**
1 Introduction . . . 318
2 Iowa Fleet data . . . 318
3 NY MTA Data . . . 319
4 Daily Stock Data: s2010 and s2011 . . . 321
5 Annual Fundamental Financial information: fnd . . . 322
6 Soap Transaction Data . . . 325

Appendix B Connecting SQL to Python or R **327**
1 Connecting to any database: ODBC and JDBC . . . 327
2 Connecting only to PostgreSQL . . . 327

Appendix C Assignments **329**
1 HW #0A: PostgreSQL Installation . . . 330
2 HW #0B: Pandas Installation . . . 331
3 HW #0C: MS CAPP Installation instructions . . . 332
4 HW #1A: Basic SQL Querying . . . 333
5 HW #1B: Basic Pandas . . . 335
6 HW #2A: Basic Functions . . . 337
7 HW #3A: Subqueries . . . 339
8 HW #3B: Subqueries in Pandas . . . 342
9 HW #4A: Aggregation . . . 345
10 HW #4B: Aggregation in Pandas . . . 347
11 HW #5A: Aggregate Functions and Dates . . . 349
12 HW #5B: Aggregate Functions and Dates . . . 351
13 HW #6A: SQL Joins (I) . . . 353
14 HW #6B: Pandas Joins (I) . . . 356
15 HW #7A: SQL Joins (II) . . . 358
16 HW #7B: Pandas Joins (II) [TBD] . . . 360
17 HW #8AO: SQL Window Functions: [TBD] . . . 362
18 HW #8A: SQL Window Functions . . . 364
19 HW #8B: Pandas Window Functions . . . 365
20 BART Project . . . 367
21 HW #5AO: Info Schema and Price-Volume Relationship [TBD] . . . 370

Appendix D Example Exams **373**

DRAFT

1 2023 CAPP Databases Final A . . . 374
2 2023 CAPP Databases Final B . . . 380
3 2023 CAPP Databases Midterm A . . . 386
4 2023 CAPP Databases Midterm B . . . 390
5 2017 SQL Final . . . 394
6 2018 SQL Final . . . 403
7 2019 Exams . . . 414
8 USF's student table . . . 433
9 FF Sales Example . . . 440
10 The Sales Rollup . . . 447
11 Sales Example I . . . 450
12 Sales Example II . . . 455

DRAFT

Introduction and Errata

DRAFT

DRAFT

Introduction & Errata

Thank you for your interest in learning Data Management via SQL and Python. The material in these lecture notes covers the vocational aspects of learning these tools in a systematic and consistent manner.

Thank you for your interest in learning SQL! At the end of this course, you will be familiar with SQL and comfortable using it in a variety of real-world situations. While we directly use PostgreSQL in the notes, nearly all of the syntax presented is compatible with alternative SQL implementations. In those cases where there are compatibility issues, we try to call them out and address them.

Neither Pandas or SQL is *difficult*, but like learning any other language it requires time and practice. The purpose of these notes and problems are not to be a readable book, but instead a set of notes which are both a reference and guide. The majority of the learning that occurs is not within the text, but within the problem sets and their solutions.

Each module within this text is designed to be a (roughly) one hour lecture. At different levels and experience it is possible for some to run short and others to run long. This course has been taught in as little as 7 weeks to masters level students and taken as long as an entire semester at the undergraduate level. Dependencies between different modules are fairly obvious and quite a bit of the more technical material can be treated as an extension (specifically Modules 4, 10, 12 and 13 and easily skip-able).

The course material is designed to be amenable to a few different environments. It has been taught at the undergraduate level, undergraduate level as well as a free-standing executive certificate. While the primary learning objectives are the same in each of these environments there are (obviously) different expectations around this course at each of these levels.

DRAFT

Undergraduate

At the undergraduate level this has been taught as a semester long course which was paired with an applied machine learning section. When teaching at the undergraduate level students were provided with access to a cloud-based relational database, with limited permissions, that contained the databases used in this course.

The coursework in these notes was paired with a group project and research paper write-up. Students were required to load their data into a database and then created a set of jupyter notebooks and Python libraries to access the data and execute on their own research plan and agenda. Groups of students then did final presentations and long-form write-ups.

Graduate

At the graduate level this has been taught as both a once-a-week, 7-week long introduction to SQL as well as a twice a week quarter length course covering both SQL and Pandas. In the former situation only

the core modules regarding SQL syntax were covered while the later included all information presented in these notes.

When teaching at the graduate level, the raw data was provided to the students with the expectations that they would be able to load it into their own SQL (local) instances and work from there. Homework problems are lightly graded and quizzes are given each week in order to assess current knowledge retention.

Executive Certificate

This course material was also taught as an Executive Certificate in a once-a-week, 3 hours per week, 7 week long format. During each week, save the first and last, students were given a short self-assessed quiz. An hour long lecture was then done followed by working on problems from the assignments with the goal to finish the "first five" for the sections covered.

When teaching in this format, only the core SQL syntax was covered and students were provided access and credentials to a cloud-based server which contained the data for the course.

DRAFT

Errata and WIPs

This document is a work in progress and contains quite a few known issues. This preface contains known issues and places where improvements are required.

Overall

1. Remove every reference to Module and change to chapter.
2. Fix the interview notes.
3. DDL
4. Categorical Data
5. Add a section to the start of the Pandas regarding "state" and how, unlike SQL, there is a current "state" of a DataFrame. E.g. row numbers matter a lot.
6. Vector DB Discussion: `https://www.ethanrosenthal.com/2023/04/10/nn-vs-ann/`

SQL

1. Rewrite `cast` section. Currently confusing.
2. Queries with out a from clause and discussion of `select 1` put at start mod 2.
3. Add to the start of the book `https://www.amazingcto.com/postgres-for-everything/`
4. Simple correlated subquery example. Current example is far too complex.
5. Rewrite NoSQL section adding information about a vector and graph databases:
   - Look at: `https://www.theregister.com/2023/03/08/great_graph_debate_wednesday/`
6. Add a resources section to the introduction which contains information on different books to consider.
   - `https://postgrespro.com/community/books/internals`
7. Online PostgresSQL explainer: `https://explain.dalibo.com/`
8. Add to performance consideration section. Discussion on the extreme case of super wide tables and how it effects performance: `https://www.cybertec-postgresql.com/en/column-order-in-postgr`
9. Add more examples of aggregation with case statements.
10. Add more formality to the discussion on what is returned and how it can be used in table vs. scalar.
11. `https://carlineng.com/?postid=sql-critique#blog`
12. Use MTA data and add hour to create timestamp in the date/time section. Add more date/time examples to date and time section, including intervals.
13. Language around analytic functions and LTV incorrect and needs to be fixed.
14. Check GSN and Zynga Dates. Add photos from Zynga as well as their MySQL solution.

Pandas

1. Add `rank` aggregation.
2. In section #1 the way that value_counts and column selection occurs is awkward. Maybe change the ordering to move value counts to after column selection.

DRAFT

3. Add reindex to index discussion in pandas. Overall discussion of muulti-index and reindex needs to be updated.
4. Move `loc` not accepting NaN to first module
5. Add example for `duplicated`
6. Date stuff needs to be redone, both as index (additional section) and as regular/type discussion.
7. MTADF comes out of nowhere in the module 2. When is it first introduced?
8. Other resources to look into:
   - Effective Pandas `https://store.metasnake.com/effective-pandas-book`
   - `https://betterprogramming.pub/pandas-illustrated-the-definitive-visual-guide-`
   - copy warning: `https://stackoverflow.com/questions/32573452/settingwithcopywarning`
9. day_name vs. weekday_name
10. `https://www.practicaldatascience.org/html/views_and_copies_in_pandas.html`
11. Add a short section on creating simple dataframes with dictionaries or lists
12. Move applymap map and apply to the next module and then take all the slice stuff and move it to the module with the groupby object.
13. Cuts / bins
14. iterrows
15. Any / All
16. Pivot
17. Add a section on analyzing the transaction data using Window Functions. Specifically, mimic the functions in the advanced joins.
18. HW #8B needs work, add more.
19. Loading and Saving Data
20. Time Series stuff
21. There are some questions in the HW about correlation, just go over this.

DRAFT

Chapter 1

Rows and Columns

DRAFT

Contents

1 What is a Relational Database 3
2 Selecting Columns 6
3 WHERE: Filtering rows 7
4 Null 9
5 ORDER BY and LIMIT 11
6 Column Numbering 16
7 Where are we: A Note on Scope 17

DRAFT

1 What is a Relational Database

- A database is a collection of any data:
  - Could be text files, a list of names, phone numbers, audio files or images.
- This is quite broad – what makes a database *Relational*?
  - Cheap Answer: A database is relational if it follows the *Relational Data Model*
  - Longer Answer: The Relational Model is a system of storing and organizing data proposed by Edgar Codd in the 1960's and 70's. Codd proposed a set of 12 rules.[1] Any database system which follows these rules has a number of important features:[2]
- Codd's 12/13 Rules:

0. **The foundation rule:** For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.
1. **The information rule:** All information in a relational data base is represented explicitly at the logical level and in exactly one way – by values in tables.
2. **The guaranteed access rule:** Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name.
3. **Systematic treatment of NULL values:** NULL values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.
4. **Dynamic online catalog based on the relational model:** The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.
5. **The comprehensive data sublanguage rule:** A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:
   - Data definition.
   - View definition.
   - Data manipulation (interactive and by program).
   - Integrity constraints.
   - Authorization.
   - Transaction boundaries (begin, commit and rollback).
6. **The view updating rule:** All views that are theoretically updatable are also updatable by the system.

DRAFT

[1]Like a good computer scientist, his rule list starts at zero, so there are actually 13 rules.
[2]These descriptions below were copied from Wikipedia, which I assume copied from the original paper.

7. **Possible for high-level insert, update, and delete:** The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data.

8. **Physical data independence:** Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.[3]

9. **Logical data independence:** Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables.[4]

10. **Integrity independence:** Integrity constraints specific to a particular relational data base must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

11. **Distribution independence:** The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only.

12. **The nonsubversion rule:** If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

- While there are many ways that these rules could be executed, the modern "Relational Database" is quite standard. On the language side, SQL is used as the primary programming language and a set of objects, discussed below, are used to conform to the rest of the rules.[5]

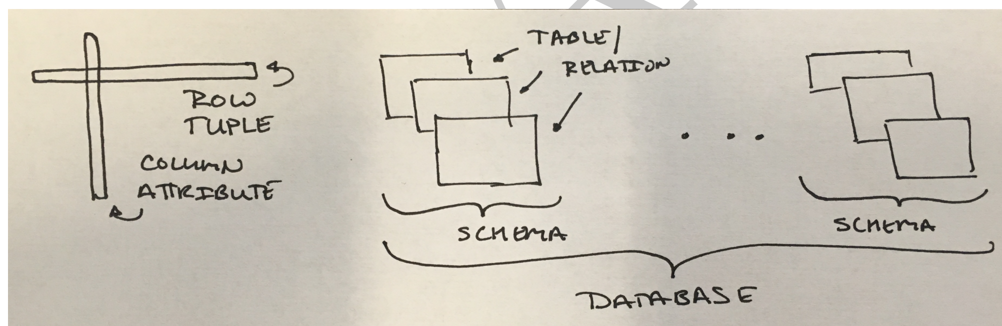


Figure 1.1: Database Objects

DRAFT

- Relational databases consist of the following objects:
  - A **tuple**, or **row**, is a single observation, like you would find in Excel.
  - An **attribute** of a tuple is a **column** and is defined by both a **name** and **type**.
  - A **relation** or **table** is a collection of rows and their attributes. It is comparable to a worksheet in Excel. Importantly, **tuples are not ordered in a relation** or, to state it alternatively, row order is arbitrary.

[3]The physical level consists of the physical storage of the data. Things like compression type, adding or removing a new hard drive or changing the directory that the database is stored in are considered "Physical Level".

[4]The logical level consists of the concept level logic sitting on-top of the physical layer. This includes adding and subtracting columns, changing an index or otherwise modifying how the information is stored without making underlying changes to the data in the database.

[5]While there is chatter about the "exact" specification of what an RDMS is and whether modern database systems actually follow the rules – that conversation is not of interest to practitioners.

  - A collection of relations is a **schema** and can be compared to an Excel workbook with multiple worksheets. Taking the Excel analogy further, just as it is straightforward to access data on different sheets, but within the same Excel file, it is straightforward to access data in a database that is within the same schema.
  - A collection of schemas is a **database**. A database can be thought of a directory full of Excel files, each with their own set of worksheets. Continuing the Excel analogy: while data *can* be accessed between different Excel files, it can be difficult; that same principle applies to databases: accessing data in two different schemas is not trivial.

- Modern relational databases use "dot" notation when referring to items in a database:

  database.schema.table.column

  Since rows are unnamed in relational databases there is no method to reference a row.

- The data stored in a relational database is strongly *typed.* Each column of data is defined as a certain *type* (think integer or string) which is defined by the operations that can act on it. For example, you can multiply numbers, but you cannot do the same to strings.

- Relational Databases work well when you have *structured* data or data with a high-degree or organization. For example, information such as "age", "first name", "last name" and "occupation" are easily stored in relational databases. Data without consistent structure, such as those collected from multiple sources or those with complex structures, such as the information contained within a personal history, tend to best stored in other structures.

- Relational databases are accessed using "Structured Query Language" ("SQL") and pronounced "Sequel".

- Importantly, relational databases are *boring.* They are designed to be warehouses of data, but NOT to do data analysis. The most common use case for relational databases is that they store information that is too large to put on a single computer. When a user or process wishes to do analysis, they connect to the database and extract only the information necessary to do their work. In other words, the database is useful for storing and accessing data, but not for doing any data science or analytics. In SQL there are no functions for "logistic regression", "random forrest", "anova" or any other common standard statistical techniques.

- Relational databases are generally *client-server* systems. In order to access data within a database the server must be up and running and a client needs to be configured to interact with it. This can happen in many different ways:
  - Both the client and server can be running on the same physical computer.
  - The server can be in the cloud and a user may be anywhere in the world.
  - The server could be in a storage room or under a desk and the client could be somewhere in the office.

- The parameters required to connect to most SQL systems include a (1) a host, (2) a username, (3) a password (4) a port to connect on and (5) the database name. The combination of these four items is referred to as a "connection string".

- When we refer to different SQL variants, such as MS-SQL, Vertica or MySQL, we are talking about the program that is run on the server, not on the client. There are clients that can connect to multiple server variants and there are clients which are specific to ones.

DRAFT

- There are many clients for SQL. One you may consider is DataGrip, by the makers of PyCharm, so if you like using that interface, use it. Others that I tend to recommend include PopSQL and Postico.[6]
- SQL is a *declarative* programming language. In a declarative language, the programmer describes the output while not explicitly detailing each step to create that output. The opposite of declarative programming is *imperative* programming which focuses on how a program should operate in a step-by-step manner.[7]

2 Selecting Columns

- **Iowa Cars:** The first dataset that we will consider is a database of car registrations from Iowa. Details about this dataset, including how to load it and its data dictionary can be found in Appendix A, Section 2.
- Let's begin with the following query template:

```
SELECT ______________  <- Comma separated List of Columns
FROM ______________ <- Table From which columns are selected from
; <- Sometimes necessary
```

- The query begins with the SELECT clause which states which columns of data we are interested in having our database return.
- The FROM clause tells the database which table to look for the data .
- Why is the ";" only "sometimes" necessary? It depends on your SQL client. The semi-colon represents the end of a statement and some SQL clients will automatically place one. The use of a semi-colon is highly recommended!
- Let's say that we have the cls.cars table structure and we want to select the "CountyName" column. We would write the following query:

```
select CountyName from cls.cars;
```

  which would return a single column and all 41,202 from the table.

- SQL ignores hard returns, additional white space and is generally case insensitive[8] so we could write the above query like this:

```
SeLeCT
        CountyNAMe                 FRom
CLS.cARS;
```

  and it would return the same data as above.

- Selecting multiple columns is straightforward:

```
select countyname, annualfee
from cls.cars;
```

[6]This site https://wiki.postgresql.org/wiki/PostgreSQL_Clients contains an up to date list of clients which are usable in PostgreSQL.
[7]Languages such as Python and R are generally considered imperative.
[8]There are a few exceptions to case insensitivity, but they are outside the scope of this class.

DRAFT

which will return two columns and all rows.

- To return **all** columns, we use an "*":

```
select * from cls.cars;
```

which will return all rows and all columns from the database.

- Be careful with the above command, SQL servers are powerful and assume you know what you are doing. If you ask for all the data from a large table... it will give you all of the data.

3 WHERE: Filtering rows

The select operator chooses which columns to return to the client while the WHERE filters out rows based on their contents.

- The WHERE syntax uses standard Boolean style logic to evaluate row-inclusion on a row-by-row basis.

- The WHERE clause occurs after the FROM clause, such as in this example:

```
select countyname, vehicletype
from cls.cars
WHERE vehicletype = 'Semi Trailer';
```

which which returns 1,683 rows.

- We can select all columns which fulfill a certain criteria:

```
select * from cls.cars where vehicletype = 'Semi Trailer';
```

which returns all columns with this vehicle type.

- Strings themselves, such as 'Semi Trailer', are case-sensitive, so the query

```
select countyname
from cls.cars
where vehicletype = 'Semi TRAiler';
```

will return zero rows.[9]

- When selecting strings we use single quotes. Double-quotes should be reserved for referencing database objects (such as tables, columns and schemas). Most databases allow users to define objects with special characters, such as spaces. We use double-quotes in these situations to refer to these objects.[10] For example, if a database had a column with a space in it, such as "user name" then to select that column would require using double quotes:

```
select "user name" from table;
```

[9] Surprisingly, this is not true for all variants of SQL. MySQL, which may be the most popular variant of SQL, is case-insensitive by default.

[10] Personally, I strongly believe that special characters should be avoided when defining database objects, but it sometimes occurs.

DRAFT

- With numbers, we can use any standard comparison operator $(=, >, <, <=, >=)$ to select rows, such as in the below,

```
select * from cls.cars where registrations > 10000;
```

returns 1,248 rows.

- There are two ways to state inequality: “$! =$” or “$<>$”.

- To combine multiple criteria we use “AND” and “OR” clauses:

```
select
    *
from
    cls.cars
where
    registrations > 1200
    and countyname = 'Wright';
```

returns 70 rows, while

```
select * from cls.cars
where
    registrations > 1200
    and registrations <= 3000
    and countyname = 'Wright';
```

returns 21 rows, or all information from Wright county where the registrations are between 1,200 and 3,000 (inclusive). A more complicated example,

```
select * from cls.cars where
    (
    (registrations > 1200 and registrations <= 3000)
    or
    (registrations > 4000 and registrations <= 4200)
    )
    and countyname = 'Wright';
```

returns 24 rows.

- Parenthesis dictate the order of operations, so this query returns all rows from Wright County with between 1,200 and 3,000 registrations or 4,000 and 4,200 registrations. To understand this query, consider the following potential values, with how they would evaluate in the above condition:

Countyname	Registrations	Yes or No
Wright	2000	Yes
Wright	4100	Yes
Right	3500	No
Wright	5000	No

4 Null

- Relational databases use *3-value* logic. Unlike some logic systems which solely consist of True and False, 3-value logic systems include a third value: Null. Null is sometimes sometimes called “unknown” or “missing”, but be careful about this phrasing as different data sources may represent those concepts separately.
- The truth tables describes how the three value logic behaves. Tables 1.2, 1.3 and 1.4 demonstrate how Null is handled in a range of situations – the key things to keep in mind is that Null does not evaluate Null, it is it’s own, separate value.

```
| AND   | TRUE   | FALSE   | NULL   |
|-------+--------+---------+--------|
| TRUE  | TRUE   | FALSE   | NULL   |
| FALSE | FALSE  | FALSE   | FALSE  |
| NULL  | NULL   | FALSE   | NULL   |
```

Figure 1.2: AND 3-value logic

```
| OR    | TRUE   | FALSE   | NULL   |
|-------+--------+---------+--------|
| TRUE  | TRUE   | TRUE    | TRUE   |
| FALSE | TRUE   | FALSE   | NULL   |
| NULL  | TRUE   | NULL    | NULL   |
```

Figure 1.3: OR 3-value logic

```
| NOT   |       |
|-------+-------|
| TRUE  | FALSE |
| FALSE | TRUE  |
| NULL  | NULL  |
```

Figure 1.4: NOT 3-value logic

- Most conditional expressions are predicated on “Success” being “True”, which means that Null *tends* to behave like “False”. For example, in a WHERE clause, if the resulting expression evaluates Null it is *not* returned. So this query, which returns Null for every row, will yield zero rows:

```
select * from cls.cars where null;
```

- Null thus represents a special class of data in the database. Consider the following query, which returns 3 rows of data:

```
SELECT
    year, registrations, annualfee
FROM
    cls.cars
WHERE
    registrations > 217000
    and year < 2010;


  year    registrations    annualfee
------  ---------------  -----------
  2006           218883
  2005           218235
  2008           217073   24160802.0
```

- The first two rows in “annualfee” are null. Let’s put a further restriction on this columns and see what happens:

```
SELECT
    year, countyname, registrations, annualfee
FROM
    cls.cars
WHERE
    registrations > 217000
    and year < 2010
    and annualfee < 25000000;


  year  countyname      registrations    annualfee
------  ------------  ---------------  -----------
  2008  Polk                   217073   24160802.0
```

This query will return 1 row (the third row from the previous query). In particular, Null annual fees are not evaluated to be true. Just to be clear, the only row that will be returned by this query, of the three that were returned will be the third row which did *not* have a Null annualfee.

- The query above only returned the non-Null annualfee rows when we put a restriction that annualfee had to be *less* than 25,000,000. Let’s put a restriction in the opposite direction, this time only returning rows *greater* than 25,000,000:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee >= 25000000
    and year < 2010;


year    countyname    registrations    annualfee
------  ------------  ---------------  -----------
```

This query returns zero rows!

- So, if we have no restriction on annualfee it returns 3 rows. If we restrict annualfee to above 25,000,000, it returns 1 row and if we restrict annualfee to below 25,000,000 then it returns 0 rows. What happened? A key relational databases feature is the following:

NULL behaves like FALSE in WHERE CLAUSES!

- In the previous example, for the first and second row the database evaluated NULL $\geq$ 25,000,000 and said that this is not true. The database also evaluated Null $<$ 25,000,000 and found it to be false. In other words, Null evaluates False in a boolean expression.

- Since Null behaves like False we need to use a different operator to compare it; this operator is `IS`:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee IS NULL
    and year < 2010;


  year  countyname      registrations  annualfee
------  ------------  ---------------  -----------
  2006  Polk                   218883
  2005  Polk                   218235
```

which will return the two rows with a Null annualfee.

- To remove NULL values we use the `IS NOT` expression:

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee is not NULL
    and year < 2010;


  year  countyname      registrations    annualfee
------  ------------  ---------------  -----------
  2008  Polk                   217073   24160802.0
```

which will return only the row with a non-Null annualfee value.

- What about this query?

```
select
    year, countyname, registrations, annualfee
from
    cls.cars
where
    registrations > 217000
    and annualfee = NULL;
```

Ha! Trick question: this will return zero rows because Null always evaluates False!

5 ORDER BY and LIMIT

- To sort the data that is returned by a query we use an ORDER BY clause:

```
select
    registrations, *
from
    cls.cars
order by registrations ASC;


  registrations    year  countyname    motorvehicle    vehiclecat    vehicletype  [...]
---------------  ------  ------------  --------------  ------------  ------------ [...]
              1    2015  Poweshiek     Yes             Truck         Tractor/Truc [...]
              1    2014  Dallas        Yes             Truck         Tractor/Truc [...]
              1    2014  Jefferson     Yes             Truck         Tractor/Truc [...]
              1    2013  Tama          Yes             Truck         Truck Tracto [...]
              1    2011  Wapello       Yes             Truck         Truck Tracto [...]
[...]
```

will return every column of data in the database ordered by registrations from low-to-high (Ascending).

- Data can also be sorted "Descending" – from high-to-low:

```
select
    registrations, countyname, year
from
    cls.cars
order by registrations DESC;


  registrations  countyname      year
---------------  ------------  ------
         218975  Polk            2015
         218883  Polk            2006
         218235  Polk            2005
         218211  Polk            2014
         217540  Polk            2016
[...]
```

Running this query will return the Null annualfee rows first! Because annualfee Nulls are sorted as if they are larger than non-Nulls, this implies that descending order returns them before non-Null values.

- ORDER BY also works with character columns:

```
select
    countyname
from
    cls.cars
order by countyname desc;


countyname
------------
Wright
Wright
Wright
Wright
Wright
[...]
```

will return the data, sorted by countyname ascending. For characters, DESC is reverse alphabetical order, while ASC is alphabetical order. PostgreSQL sorts character data in a case insensitive manner.

- Under the default sort order (Ascending), Null values are *last*, which means that they are treated as if (a) they are larger than non-Nulls, when numbers and (b) alphabetically last.
- The default sort order, if neither ASC or DESC is specified is ASC.
- Multiple sort orders can be combined:

```
select
    countyname, year
from
    cls.cars
order by countyname desc, year asc;


countyname      year
------------  ------
Wright          2005
Wright          2005
Wright          2005
Wright          2005
Wright          2005
[...]
```

will return data sorted by countyname first, and then within common countyname's sorted by year (ascending).

- In order to limit the number of rows returned to the client we use the LIMIT command. For example, the following query:

```
select
    *
from
    cls.cars
limit 10;


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage      [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                          [...]
  2011  Jasper        Yes             Moped         Moped                        [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons       [...]
  2015  Palo Alto     No              Trailer       Travel Trailer               [...]
  2016  Adair         Yes             Truck         Truck           3 Tons       [...]
[...]
```

DRAFT

returns every column in the cars table, but only 10 rows. Note that this is **not** the first 10 rows, it is an *arbitrary* 10 rows. If you run the query again you may get a different set of rows!

- We can combine LIMIT with the rest of our commands:

```
select
    registrations, annualfee
from
    cls.cars
where
    countyname = 'Scott'
limit 100;


  registrations    annualfee
---------------  -----------
          31626    1700000.0
              3        376.0
           1779     553072.0
            172      40480.0
           2972     135942.0
[...]
```

will return 100 rows of data. Note that limit is evaluated after the where command, so as long as there are 100 rows with countyname = 'Scott', this will return 100 rows.

The LIMIT clause is not present in all versions of SQL. Other SQL variants use alternative syntax to limit the rows returned. The following table highlights those differences and presents a query using each version:

Variant	Syntax	Example
MySql	LIMIT	select * from cls.cars order by annualfee asc limit 10;
MS-SQL	TOP	select top 10 * from cls.cars order by annualfee asc;
Older Oracle	rownum	select * from (select * from cls.cars order by annualfee asc) where rownum <= 10;

- A common use case for ORDER BY and LIMIT is to return the top X of something, like the query below, which returns the five largest registration values:

```
select
    registrations
from
    cls.cars
order by registrations desc
limit 5;


  registrations
---------------
         218975
         218883
         218235
         218211
         217540
```

- Some other examples: Top 15 rows ordered by registrations:

```
select
    *
from
    cls.cars
order by registrations desc
limit 15;


  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2015  Polk          Yes             Automobile    Automobile                    [...]
  2006  Polk          Yes             Automobile    Automobile                    [...]
  2005  Polk          Yes             Automobile    Automobile                    [...]
  2014  Polk          Yes             Automobile    Automobile                    [...]
  2016  Polk          Yes             Automobile    Automobile                    [...]
[...]
```

- Top-15 rows, countyname only, where annualfee is not equal to zero.

```
select countyname
from cls.cars
where annualfee <> 0
order by registrations desc limit 15;


countyname
------------
Polk
Polk
Polk
Polk
Polk
[...]
```

This query demonstrates that (1) we can order by columns that are not selected and (2) we can remove both NULL and zero annualfees.[11]

6 Column Numbering

- In this section we introduce an SQL feature that make your life easier: using column number order in an ORDER BY clause.
- Instead of putting the column name, we can use the column number to specify how we should order the rows:

```
select
    registrations
    , countyname
from
    cls.cars
where
    annualfee <> 0
order by 1 desc
limit 15;


  registrations  countyname
---------------  ------------
         218975  Polk
         218211  Polk
         217540  Polk
         217073  Polk
         216792  Polk
[...]
```

In the query above the “1” in the ORDER BY refers not to the number 1, but to the first column position, which in this case is “registrations”. We can also use ASC, DESC and multiple sort orders using this notation:

[11] Since NULL <> is always false this removes all the null annualfee values.

```
select
    registrations
    , countyname
from
    cls.cars
where
    annualfee <> 0
order by 2, 1 desc;


  registrations  countyname
---------------  ------------
           4247  Adair
           4137  Adair
           4109  Adair
           3916  Adair
           3768  Adair
[...]
```

The data from this example is sorted by countyname first and then, within each county sorted by the number of registrations. There are only a few operators that allow for column numbering. The ORDER BY operator is one of them.

- The downside of using column numbering syntax is that if you change the select statement, such as adding or subtracting a column the query will not raise an error, but will no longer return the same data. For this reason a number of organizations avoid using this syntax and require full column name expressions at all times.

7 Where are we: A Note on Scope

- A common issue that new SQL users have is how much detail to include when writing a query. For example, if we assume that the database name is "sql_class", then the following two queries will return the same data, though they look very different:

```
select sql_class.cls.cars.year from cls.cars;
```

and

```
select year from cls.cars;
```

- We call this issue *scope*. Scope is the region of a query where a particular name is valid. The standard way that we write queries is to include the schema and table in the FROM clause, but only include the column names in the rest of the query, unless those column names are not fully specified.
- Note that the table was written as "cars" and not "cls.cars" or something even more specific. This may return an error for you depending on how your "search path" is set. The search path represents where the database looks, by default, for tables. If you are working with data that is not in your search path, you will need to include the schema name, otherwise the database will return an error. In the notes for this class we will ignore issues surrounding this.[12]

DRAFT

[12]More information here: `https://www.postgresql.org/docs/current/static/ddl-schemas.html`

DRAFT

Chapter 2

Basic Manipulations

DRAFT

Contents

1 Types **21**
2 Renaming a Column **23**
3 Basic Mathematical Manipulations, ABS and LEAST/GREATEST **24**
4 Queries without a FROM Clause and Singletons **28**
5 String Functions: LEFT, RIGHT, LOWER, UPPER, LENGTH, TRIM and CONCAT . **29**
6 ROUND and Changing Types (CAST) **33**
7 CAST and changing types **33**

DRAFT

Up to this point we have refrained from transforming any of the data that is being returned in our queries. In this module we being working on manipulating the data that is being returned via functions, renaming and other methods. Importantly, none of what we are doing changes the underlying data; it simply transforms what is being returned to the client.

Before manipulating, however, we need to understand data within a relational database and how it is represented. In particular, we need to understand "types".

1 Types

- Relational databases are "strongly" typed, meaning that there are strict rules around what operations can be performed on what data.
- As in other computer languages, types determines both what operations are available and how operations behave as a function of the data contained therein.
- In Relational Databases, columns are typed and set when a table is created. A column can only be a single type.
- Relational databases support a variety of different data types. In this section we will discuss the most commonly used ones, a hierarchy of which can be found in Figure 2.1.

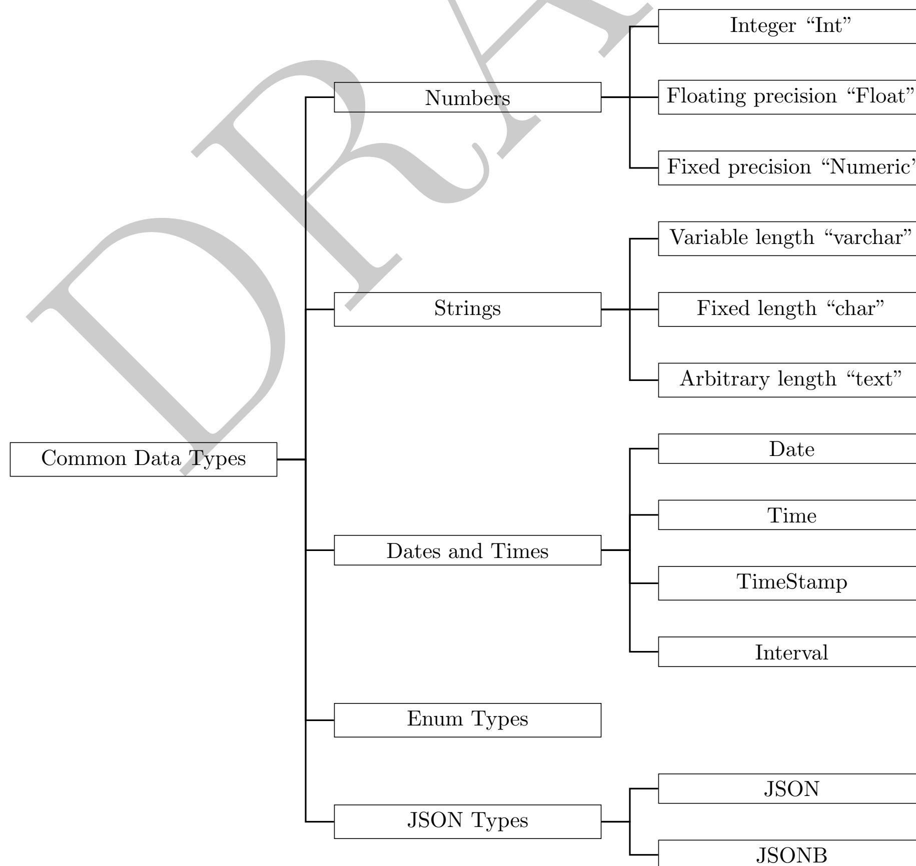


Figure 2.1: Common relational database data types

DRAFT

Numbers

There are three "styles" of numbers:

1. **Integer:** These are whole numbers and there are actually 3 different types: smallint (2 bytes, can store -32,768 to +32,767 (2^{15})), int (4 bytes, can store -2,147,483,648 to +2,147,483,647 (2^{31})) and bigint (8 bytes, can store -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (2^{63})).
2. **Float:** A floating point number is an inexact, variable precision numeric type, usually coming in two flavors: real (4 bytes, 1E-37 to 1E+37 with a precision of at least 6 decimal digits) and double (8 bytes, 1E-307 to 1E+308 with a precision of at least 15 digit).
3. **Numeric:** A numeric has a user-defined fixed precision (like 2 decimal places). They vary in size and type depending on the amount of precision required. An example use of fixed precision is storing information about money; there is a fixed cut-off (penny) of precision.

In practice database administrators tend to stick to using integers and floats with an occasional numeric types.

Strings

The three most common string types used are:

1. **Variable length:** The "varchar" type is used for variable length strings, but with a maximum number of specified characters. For example, a varchar(10) can contain any string, as long as the number of characters is less than or equal to 10.
2. **Fixed length:** The "char" type is used for fixed length strings. For example, a char(10) can contain any string, as long as the number of characters is less than or equal to 10. The difference between char and varchar is that this type always reserves space for additional characters, up to the the max, while a varchar does not. So, to store the names "Nick", "John" and "Reggie" as a varchar(6) would take (approximately) ($4 + 4 + 6 = 16$ bytes) while storing those same names as a char(6) would take (approximately) $6 + 6 + 6 = 18$ bytes.
3. **Arbitrary length:** The "text" type (sometimes called blob) is used for strings of arbitrary length. For example, if you wanted to store yelp comments you would use a text field, since the comments can be any length. Text fields are generally avoided when another type can be used due to storage efficiency.

Enum

- For categorical data databases use what is called an "enum" or "enumerated" type.
- This type stores the data as an integer which also has a "map" that maps those numbers to specific values.
- The classic version of this is gender. Consider a survey with the following options:

	No. Chars	Enum Value
Woman	5	1
Man	3	2
Transgender	11	3
Non-binary/non-conforming	25	4
Prefer not to respond	21	5

- In this example, storing the data as an enum would save a ton of space over storing it as text.

DRAFT

- The downside is increased complexity and issues with comparisons (do you compare based on the map or on the text value)?
- All modern databases have a version of this, but we won't get too much into the details in this course.

JSON

- Modern databases usually have two different options for storing JSON information: a raw representation and a binary representation.
- The raw representation is just a text blob that, by calling it "JSON" you get access to special functions only available to JSON objects, specifically functions around keys and values.
- The JSONB representation is a further parsed, binary representation of the JSON data. JSONB data (usually) is slower to load into the database due to the additional type conversions, but faster to do lookup operations on.
- All modern databases support JSON path (sometimes called JSONpath) syntax for accessing operators. This will be discussed later.

Dates

We will hold off on discussing dates until Module 6.

2 Renaming a Column

- The first thing we will learn to do is change the name of tables and columns that are being returned.
- We sometimes want to rename columns. To do this, we use the `AS` operator:

```
select
    registrations as reg2
from
    cls.cars;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

  The query above will return a single column, with the name `reg2`.

- We can also use it to rename tables, though this won't be useful for a few weeks!

DRAFT

```
select
    registrations as reg2
from
    cls.cars as c2;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

- We can actually just skip the AS completely, thought it isn't recommended since it can make the query more difficult to read.

```
select
    registrations  reg2
from
    cls.cars  c2;


  reg2
------
     5
   198
  5020
   366
  2507
[...]
```

3 Basic Mathematical Manipulations, ABS and LEAST/GREATEST

- If, instead of selecting a column directly from the table, we put down a single value, then that value will be repeated for each row returned.
- For example, consider the following query:

```
select
    1 as v1, 2 as v2, 'Nick' as name, vehicletype
from
    cls.cars;


  v1    v2  name    vehicletype
----  ----  ------  --------------
   1     2  Nick    Bus
   1     2  Nick    Moped
   1     2  Nick    Truck
   1     2  Nick    Travel Trailer
   1     2  Nick    Truck
[...]
```

- Note that the data is repeated once for each row and no rows are being generated.
- We can also manipulate the data that is being returned on a row-by-row basis by using functions within the select.
- For example, we can do basic math functions:

```
select
    registrations + 10 as reg2
    , registrations
from
    cls.cars


  reg2    registrations
------  ---------------
    15                5
   208              198
  5030             5020
   376              366
  2517             2507
[...]
```

Note that what this does is create a synthetic column of the number ten (repeated for each row) and then adds that to the column "reg". The result is that each entry in "reg2" is equal to "reg" plus 10.

- All standard mathematical operations $(+, -, /, *)$ are all supported and math can be done between columns, such as:

```
select
    registrations + 10 as reg2
    , annualfee * annualfee as annualfee_sq
    , registrations
from
    cls.cars;


  reg2      annualfee_sq    registrations
------  ----------------  ---------------
    15  462400                          5
   208       1.921e+06                198
  5030       9.60083e+10             5020
   376       3.39039e+08              366
  2517       1.7873e+10              2507
[...]
```

- What if we fail to rename the column with AS? The database will generate a column name for us:

```
select
    registrations + 10
    , annualfee * annualfee
    , registrations
from
    cls.cars;


  ?column?          ?column?    registrations
----------  ----------------  ---------------
        15  462400                          5
       208       1.921e+06                198
      5030       9.60083e+10             5020
       376       3.39039e+08              366
      2517       1.7873e+10              2507
[...]
```

In this case the database has no idea what to name the column so calls it ?column?.

- SQL also has more advanced functions, many of which are similar to Excel. For example, the absolute value function (ABS), which returns the magnitude of a number without regard for its sign, can be used to return a modified column:

```
SELECT
    abs( registrations - 1000 ) as abs_reg, registrations
FROM
    cls.cars;


  abs_reg    registrations
---------  ---------------
      995                5
      802              198
     4020             5020
      634              366
     1507             2507
[...]
```

returns two columns from cars. The first is the absolute value of 1,000 subtracted from registrations and the second is the registrations number.

- As with the other SQL functions we have seen, these can be used within a `WHERE` clause:

```
SELECT
    *
FROM
    cls.cars
WHERE
    abs(registrations - 1000) <= 20;


  year  countyname    motorvehicle    vehiclecat    vehicletype      tonnage      [...]
------  ------------  --------------  ------------  ---------------  ---------  - [...]
  2005  Hardin        Yes             Motorcycle    Motorcycle                    [...]
  2013  Mahaska       No              Trailer       Regular Trailer               [...]
  2008  Boone         No              Trailer       Travel Trailer                [...]
  2016  Marion        No              Trailer       Semi Trailer                  [...]
  2012  Webster       No              Trailer       Semi Trailer                  [...]
[...]
```

which returns the 231 rows where the number of registrations are between 980 and 1,020.

- The functions `LEAST` and `GREATEST` do exactly what they say – they return the highest and lowest value in a particular set of observations. Note that `LEAST` and `GREATEST` only work within a single row:

DRAFT

```
select
    countyname
    , abs(registrations - 100) as c1
    , abs(registrations - 30) as c2
    , registrations
    , least( abs(registrations - 100), abs(registrations - 30)) as calc_1
    , greatest( abs(registrations - 100), abs(registrations - 30)) as calc_2
from cls.cars
where registrations >= 64 and registrations <= 66
and countyname = 'Wright';


countyname      c1    c2    registrations    calc_1    calc_2
------------  ----  ----  ---------------  --------  --------
Wright          36    34               64        34        36
Wright          34    36               66        34        36
Wright          35    35               65        35        35
Wright          36    34               64        34        36
Wright          36    34               64        34        36
[...]
```

4 Queries without a FROM Clause and Singletons

- SQL allows for queries without a FROM. When doing this, no columns can be referenced, but the query will be executed as a single expression. This is handy when running tests, such as if we didn't understand the ABS function:

```
select abs( -5 ) as calc;


  calc
------
     5
```

We can do this with almost any SQL function, including mathematical operations:

```
select 5 * 10 as calc;


  calc
------
    50
```

- If a query returns a single value, which we will call a *singleton* in this class, then we can treat that value as what it returns. The code below, for example, returns twice the largest registrations:

```
 select 2 * (select registrations from cls.cars order by 1 desc limit 1) as calc;


  calc
------
437950
```

- We can also use this in a WHERE clause:

```
select registrations
from
    cls.cars
where
    registrations >
        10*(select registrations from cls.cars order by 1 asc limit 1)
order by registrations asc
limit 10;


  registrations
---------------
             11
             11
             11
             11
             11
[...]
```

This query will return the registrations in cls.cars which are 10 times more than the smallest value. It will only return the smallest 10 of those rows.

5 String Functions: LEFT, RIGHT, LOWER, UPPER, LENGTH, TRIM and CONCAT

- The string operators LEFT and RIGHT behave just as in Excel: they take the left or right characters of a string. For example:

```
select left( 'THIS STRING', 4) as left_4;


left_4
--------
THIS
```

will return 'THIS' since it is the four left most letters of the string in question.

- Both the LEFT and RIGHT commands take the same inputs: a string and the number of characters to cut:

```
select
    countyname
    , left( countyname, 4) as left_4
    , right( countyname, 4) as right_4
from
    cls.cars;


countyname    left_4    right_4
------------  --------  ---------
Ida           Ida       Ida
Jasper        Jasp      sper
Harrison      Harr      ison
Palo Alto     Palo      Alto
Adair         Adai      dair
[...]
```

- Two other string functions that behave similarly to Excel are `LOWER` and `UPPER`, which return a lowercase and uppercase version of a string column:

```
select
    countyname
    , lower(countyname) as lc
    , upper(countyname) as uc
from
    cls.cars;


countyname    lc         uc
------------  ---------  ---------
Ida           ida        IDA
Jasper        jasper     JASPER
Harrison      harrison   HARRISON
Palo Alto     palo alto  PALO ALTO
Adair         adair      ADAIR
[...]
```

will return the countyname (with capital casing, such as "Adair") and also a lower- and upper-case version of the countyname.

- Note that we can nest functions:

```
select
    lower( left(countyname, 4)) as ll4
    ,countyname
from
    cls.cars;


ll4    countyname
-----  ------------
ida    Ida
jasp   Jasper
harr   Harrison
palo   Palo Alto
adai   Adair
[...]
```

- the LENGTH command returns the length of a string.[1] For example:

```
select
    countyname, length(countyname) as len
from
    cls.cars;


countyname      len
------------  -----
Ida               3
Jasper            6
Harrison          8
Palo Alto         9
Adair             5
[...]
```

- The `TRIM` command can be used to remove letters from a string. The default behavior is to remove spaces, but it is possible to use it for other things.

```
 select '     aaaa' as val1, trim('     aaa     ') as trm;


val1    trm
------  -----
aaaa    aaa
```

Importantly the leading and trailing spaces have been removed from the string. Note that the commands `LTRIM` and `RTRIM` do what they are expected to do – trim from only a single side.

- To put two strings together, similar to an "&" in Excel, we can use a concatenation operator, "||". For example, the following query will return a single column with the countyname twice.

[1]In MS-SQL this is LEN, not LENGTH.

```
select
    countyname || countyname as str_calc
from
    cls.cars;


str_calc
------------------
IdaIda
JasperJasper
HarrisonHarrison
Palo AltoPalo Alto
AdairAdair
[...]
```

- We can also put a constant into the string concatenation to modify it, as in the following example:

```
select
    'County Name = ' || countyname as str_calc
from
    cls.cars;


str_calc
-----------------------
County Name = Ida
County Name = Jasper
County Name = Harrison
County Name = Palo Alto
County Name = Adair
[...]
```

DRAFT

Different variants of SQL use different operators for string concatenation:

SQL Variant	Syntax	Example
MySQL	concat()	select concat(col1, col2) from tablename;
MS-SQL	+	select col1 + col2 from tablename;

- A final useful command for parsing strings is `LENGTH`, which returns the length of a string. We can use this with right and left to uppercase the last letter of a string:

```
select
    left(countyname, length(countyname) - 1)
        || upper( right( countyname, 1) ) as lastUpper
from
    cls.cars;


lastupper
-----------
IdA
JaspeR
HarrisoN
Palo AltO
AdaiR
[...]
```

which returns a list of countynames with both the first and last letter upper-cased!

6 ROUND and Changing Types (CAST)

7 CAST and changing types

- It can be the case that you want to switch data types and then do operations on them.
- To do this we use the "CAST" operator, which takes a column and a target data type as its inputs. Unlike other functions, however, the word "as" is used to split the inputs. Consider the following examples:

```
select 125.5 + 4 as ans;


  ans
-----
129.5
```

```
select '125.5' + 4 as ans;
ERROR:  invalid input syntax for integer: "125.5"
LINE 1: select '125.5' + 4 as ans;
^
```

```
select cast( '125.5' as float) + 4 as ans;


  ans
-----
129.5
```

The first query returns the expected answer while the second errors out because it tries to add a string and an integer. The third query uses the cast operator to change the data type.

- Rather than using `CAST` PostgreSQL provides a double colon operator to do the same thing:

```
select '123.5'::float + 4 as ans;


  ans
-----
127.5
```

- Finally, keep in mind that PostgreSQL will attempt to do many conversions, even if you don't explicitly specify them. For example:

```
select '123' + 4 as ans;


  ans
-----
  127
```

Surprisingly, the database is able to make this conversion and thus does the math correctly.

- One commonly used function is the `ROUND` command which rounds a number.
- Lets say that we wanted to get the `annualfee` and `registrations` rounded to the nearest 100. In this case we could start by doing the following:

```
select
    round( registrations, -2 ) as rounded_reg
    , registrations
from
    cls.cars;


  rounded_reg    registrations
-------------  ---------------
            0                5
          200              198
         5000             5020
          400              366
         2500             2507
[...]
```

As from the results above, the `ROUND` commands rounds numbers to the place specified in the integer following the value to be rounded. In this example the rounding occurs to the -2 position which is the hundreds place.

Moving to `annualfee`, we could write it as:

```
select
    round( annualfee, -2 ) as rounded_af
    , annualfee
from
    cls.cars;
```

which would return:

```
ERROR:  function round(double precision, integer) does not exist
LINE 4:     , round( annualfee, -2 ) as rounded_af
              ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
```

Why does this return an error?!?!

The round command is *type* dependent. If you have an integer or a numeric type, the syntax is `ROUND( column, integer )` where the integer determines where to round the value. If the integer is positive, it will round to values *after* the decimal while negative integers in the `ROUND` will return values rounded to places before the decimal, as in the example above. On the other hand, floats (called "double precision" in the error) do not accept a second argument and will only round to the nearest integer!

- So how do we handle rounding to the nearest hundreds for a float? There are two options: we either transform the column to use the `ROUND` command on floats or we `CAST` the float as a different type (either numeric or integer) and then use the available parameters therein.
- The first option:

```
select
    round(registrations, -2) as rounded_reg
   , registrations
   , 100*round( annualfee/100) as rounded_af
   , annualfee
from
    cls.cars;


 rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
           0                5           700          680
         200              198          1400         1386
        5000             5020        309900       309852
         400              366         18400        18413
        2500             2507        133700       133690
[...]
```

In the example above the column `annualfee/100` is a floating point type which does not take an additional argument in the function and instead just rounds to the nearest whole number. Since it's been divided by 100, this will return the number rounded to the nearest 100. We then multiply it against 100 to get the original scale.

- The second option is to cast, or change the variables type, in a few different ways:
    1. **Use the CAST function** We can use the `CAST` command in order to explicitly change the

DRAFT

type. The CAST command is a bit awkward syntactically, as can be seen below:

```
select
    round(registrations, -2) as rounded_reg
   , registrations
   , round( cast( annualfee as int), -2) as rounded_af
   , annualfee
from
    cls.cars;


  rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
            0                5           700          680
          200              198          1400         1386
         5000             5020        309900       309852
          400              366         18400        18413
         2500             2507        133700       133690
[...]
```

rather than using standard parameters, a more sentence like construction occurs.

2. **Conversion with ::** We can use :: to explicitly cast a variable from one type to another. **This is Postgres only!**

```
select
    round(registrations, -2) as rounded_reg
   , registrations
   , round( annualfee::int, -2) as rounded_af
   , annualfee
from
    cls.cars;


  rounded_reg    registrations    rounded_af    annualfee
-------------  ---------------  ------------  -----------
            0                5           700          680
          200              198          1400         1386
         5000             5020        309900       309852
          400              366         18400        18413
         2500             2507        133700       133690
[...]
```

3. **Implicit conversion:** While not possible in every situation, Postgres will implicitly convert between types when operators are applied. For example, if you multiply a float against an integer, the result will be a float:

```
select
    annualfee / 5.0 as af
from
    cls.cars;


     af
-------
  136
  277.2
61970.4
 3682.6
26738
[...]
```

In this case, `annualfee` has been converted from an integer to a float.

- There is a big "gotcha" when using implicit conversion – when doing it the database attempts to determine which type you want if you aren't careful you may end with an unanticipated result. Consider the following:
- Look at what following returns, given that there are 659 rows where registrations is equal to 5:

```
select registrations / 10 as calc
from cls.cars where registrations = 5;


  calc
------
     0
     0
     0
     0
     0
[...]
```

- Why is this occurring? Because the database sees a query which divides two integers and thus assumes that the result is also going to be an integer. Importantly – this isn't rounding, it is simply cutting off the value.
- We can use implicit methods of conversion in order to solve this. Consider the following:

```
select registrations *1.0 / 10 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

or

```
select registrations /10.0 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

or

```
select registrations::float /10 as calc
from cls.cars where registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

The first two solutions work because they introduce a number with a decimal component. When the database attempts to do math between decimals and integers it presumes that the answer is going to be decimal and we get the expected result. The third answer uses the "::" operator to convert the integer into a floating point number.

There is one important difference between the first two solutions and the final solution. The final solution converts the data into a floating point number, not a numeric type. As we will learn later, these are not equivalent and there can be strong reasons to prefer one data type over the other.

- Finally, we could use the CAST function in order to complete this operation:

```
select
    CAST( registrations as float)/10 as calc
from
    cls.cars
where
    registrations = 5;


  calc
------
   0.5
   0.5
   0.5
   0.5
   0.5
[...]
```

DRAFT

DRAFT

Chapter 3

Subqueries, Distinct & Case

DRAFT

Contents

1 Query Evaluation Order: SELECT and WHERE **43**

2 Comparisons: BETWEEN, LIKE and ILIKE **45**

3 CASE: Conditional Logic **47**

4 The DISTINCT Operator **52**

5 Subqueries (IN, ANY, ALL) **55**

6 Correlated Subqueries **58**

DRAFT

1 Query Evaluation Order: SELECT and WHERE

- Assume that we wanted to look at the registrations per dollar of annualfee for 4 Ton Truck Tractors in Scott county. We could start with the following query:

```
select
    year, registrations, annualfee
from
    cls.cars
where
    countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';


  year    registrations    annualfee
------  ---------------  -----------
  2010                1            0
  2009                1            0
  2007                1            5
  2008                2           85
```

- To determine the registrations per annual fee, we could change the select statement to the following:

```
select
    year, registrations::float/ annualfee as ratio
from
    cls.cars
where
    countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';
```

which yields an error:

```
ERROR:  division by zero
```

Unsurprisingly, rows with an annualfee equal to zero are causing this query to fail.

- To handle this we can remove those rows that cause this query to fail:

```
select
    year, registrations::float/ annualfee as ratio
from
    cls.cars
where
    annualfee > 0
    and countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';


  year      ratio
------  ---------
  2007  0.2
  2008  0.0235294
```

which will return only the two rows where the division by zero is not an issue. Notice about this query is that the `WHERE` clause is evaluated *before* the `SELECT` statement is evaluated.

- This allows the user to exclude observations that may generate problems *before* the `SELECT` statement operates on them.
- An implication of this is that since `SELECT` is done *after* `WHERE`, things defined in the `SELECT` are *not* available in the `WHERE`:

```
SELECT
    year, annualfee::float / registrations as avg_fee
from
    cls.cars
where avg_fee > 0;
```

```
ERROR:  column "avg_fee" does not exist
```

Why did this happen? It happened because the column `avg_fee` isn't defined at the time that the `WHERE` clause is executed.

- The same logic applies to the `FROM` clause, which is evaluated *first*. Consider the following query, which renames our table into something else.

```
select
        renamed_table.*
from
        cls.cars as renamed_table
limit 100;


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                           [...]
  2011  Jasper        Yes             Moped         Moped                         [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons        [...]
  2015  Palo Alto     No              Trailer       Travel Trailer                [...]
  2016  Adair         Yes             Truck         Truck           3 Tons        [...]
[...]
```

DRAFT

In this query the table has been renamed in the `FROM` clause and that naming is passed through to the `SELECT` statement. If we were to instead try to reference `cls.cars` in the `SELECT` after the renaming, an error will occur:

```
select
        cars.*
from
        cls.cars as renamed_table
limit 100;
```

```
ERROR:  missing FROM-clause entry for table "cars"
LINE 2:  cars.*
```

Once again this confirms that the `FROM` clause is evaluated *before* `SELECT`.

2 Comparisons: BETWEEN, LIKE and ILIKE

- Another common comparison operator is BETWEEN:

```
select
    *
from
    cls.cars
where
    registrations between 2050 and 2100;


  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2011  Monroe        Yes             Multi-purpose  Multi-purpose                [...]
  2010  Shelby        No              Trailer        Small Regular Trailer        [...]
  2015  Ida           Yes             Truck          Truck                  3 Ton [...]
  2013  Dubuque       Yes             Truck          Truck                  6+ To [...]
  2010  Woodbury      No              Trailer        Semi Trailer                 [...]
[...]
```

will return all columns from the table cars where registrations are between 2050 and 2100. Note that this is equivalent to:

```
select
    *
from
    cls.cars
where
    registrations >= 2050 and 2100 >= registrations;


  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2011  Monroe        Yes             Multi-purpose  Multi-purpose                [...]
  2010  Shelby        No              Trailer        Small Regular Trailer        [...]
  2015  Ida           Yes             Truck          Truck                  3 Ton [...]
  2013  Dubuque       Yes             Truck          Truck                  6+ To [...]
  2010  Woodbury      No              Trailer        Semi Trailer                 [...]
[...]
```

In other words, BETWEEN is inclusive as it includes both end points.

- BETWEEN can also be used with strings, but be careful when doing so. In our cars database, for example, there is a single county that begins with the letter 'R' ("Ringgold"). If you run the following query:

```
select
    *
from
    cls.cars
where
    countyname between 'R' and 'R';


year    countyname    motorvehicle    vehiclecat    vehicletype    tonnage    reg [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
```

will return zero rows! BETWEEN is computed using alphabetical order and, since "R" is before "Ringgold", alphabetically, this means that it won't be returned by this query. Instead, the following query will return all rows with a countyname which begins with the letter 'R':

```
select
    *
from
    cls.cars
where
    countyname between 'R' and 'S';


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ------------- [...]
  2011  Ringgold      Yes             Bus           Bus                           [...]
  2014  Ringgold      Yes             Truck         Truck           6+ Tons Non-S [...]
  2016  Ringgold      Yes             Moped         Moped                         [...]
  2011  Ringgold      Yes             Motorcycle    Motorcycle                    [...]
  2005  Ringgold      Yes             Motor Home    Motor Home - B                [...]
[...]
```

- Second note: alphabetical order PostgreSQL is case insensitive. If you sort the following data:

```
A B D E c f g h
```

the result will be:

```
A B c D E f g h
```

- To further match strings we can use LIKE and ILIKE which searches for specified patterns within a string. Using LIKE without any special characters yields a simple equality comparison:

```
where countyname like 'Ringgold'
```

is equivalent to:

```
where countyname = 'Ringgold'
```

- ILIKE on the other hand is a case insensitive matching. In other words, the following where clauses

DRAFT

will return all rows from Ringgold county:

```
where countyname ilike 'ringgold'

where countyname ilike 'RINGgold'
```

- Both like and ilike allow for more complex pattern matching using percent sign ("%") and underscore ("_"). The percent sign is used to match any string while the underscore matches a single character. We call these types of characters "wildcards" and they allow users to create more complex matching criteria. Continuing with the example of the county of "Ringgold":

Clause	Will Match Ringgold?
like '%inggold'	Yes
like 'ring_old'	No
ilike 'ring_old'	Yes
like 'r%'	No
ilike 'r%'	Yes
ilike '%ringgold%'	Yes

- Remember that Null presents as False, even with wildcard characters. If there was a column in a table called "alwaysNull" which was Null in every row, the following:

```
where alwaysNull ilike '%'
```

would return zero rows.

- One difference between % and _ is that underscore *requires* a character to be there. For example, the string '_Ringgold' will **not** match Ringgold while '%Ringgold' will match.
- **Performance considerations:** Be mindful when using `LIKE` and `ILIKE` as they are expensive for the database to evaluate. When evaluating these expressions, the database moves from the first to last character within each string attempting to determine if each row matches the criteria. Whenever possible, minimize the use of wildcard characters.

3 CASE: Conditional Logic

- We have covered how to use a `SELECT` statement to manipulate columns. For example, we can easily add numbers together or transform a string. An extension of this is to change columns conditionally. To do this we use the `CASE` statement, which allows us to conditionally transform what the database returns.
- In the Iowa cars data we may be interested in doing analysis comparing those rows with more than 100 registrations against those with less than 100 registrations. As an example, consider the following query:

DRAFT

```
SELECT
    CASE
        WHEN registrations > 100 THEN 'BIG'
        ELSE 'SMALL'
    END as regSize
    , *
from
    cls.cars;


regsize      year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
SMALL        2008  Ida           Yes             Bus           Bus                [...]
BIG          2011  Jasper        Yes             Moped         Moped              [...]
BIG          2012  Harrison      Yes             Truck         Truck            3 [...]
BIG          2015  Palo Alto     No              Trailer       Travel Trailer     [...]
BIG          2016  Adair         Yes             Truck         Truck            3 [...]
[...]
```

This query will return all the columns in the database and one more column, with the name "regSize" that takes the value of "BIG" or "SMALL" depending on if the number of registrations is greater than 100.

- In the case of a Null value for registration it would fail the initial conditional and then be caught by the
- The `ELSE` clause is optional. The query below provides an example without an `ELSE` clause:

```
select
    case
        WHEN registrations > 100 then 'BIG'
    END as regsize
from
    cls.cars;


regsize
---------

BIG
BIG
BIG
BIG
[...]
```

In this case, the column regsize will have the value 'BIG' for registrations greater than 100. For values of registration less than 100, the value in the column will be Null.

- The `CASE` statement is evaluated row-by-row.
- We can add additional criteria by using multiple `WHEN` arguments. For example, we may want to do analysis on four different size criteria as can be seen in this query:

DRAFT

```
SELECT
    CASE
        WHEN registrations > 1000 THEN 'VERY VERY BIG'
        WHEN registrations > 500 THEN 'VERY BIG'
        WHEN registrations > 100 THEN 'BIG'
        ELSE 'SMALL'
    END as regSize
    , *
from
     cls.cars;


regsize          year  countyname    motorvehicle    vehiclecat    vehicletype    [...]
-------------  ------  ------------  --------------  ------------  -------------- [...]
SMALL            2008  Ida           Yes             Bus           Bus            [...]
BIG              2011  Jasper        Yes             Moped         Moped          [...]
VERY VERY BIG    2012  Harrison      Yes             Truck         Truck          [...]
BIG              2015  Palo Alto     No              Trailer       Travel Trailer [...]
VERY VERY BIG    2016  Adair         Yes             Truck         Truck          [...]
[...]
```

We only needed to include ">" signs because each of our inequalities excludes the previous. In other words, when the database evaluates the above it checks the `WHEN` statements in order: it first checks to determine if the number of registrations is greater than 1000, then if it is greater than 500, then if it is greater than 100 and finally, only if all 3 of those criteria fail, will it assign the value of "SMALL".

If the query was written this way:

```
SELECT
    CASE
        WHEN registrations > 500 THEN 'VERY BIG'
        WHEN registrations > 1000 THEN 'VERY VERY BIG'
        WHEN registrations > 100 THEN 'BIG'
    ELSE 'SMALL'
    END as regSize
    , *
from
    cls.cars;


regsize      year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
SMALL        2008  Ida           Yes             Bus           Bus                [...]
BIG          2011  Jasper        Yes             Moped         Moped              [...]
VERY BIG     2012  Harrison      Yes             Truck         Truck           3  [...]
BIG          2015  Palo Alto     No              Trailer       Travel Trailer     [...]
VERY BIG     2016  Adair         Yes             Truck         Truck           3  [...]
[...]
```

then *zero* observations would be classified as "VERY VERY BIG" since every row with registrations greater than 1000 are also greater than 500.

- When using case statements we can use any statement that we would use in a `WHERE` clause, including using `AND` and `OR` to create more complex Boolean statements:

```
select
    case
        when registrations > 500 and annualfee > 500 THEN 'Type 1'
        when registrations >= 500 and annualfee < 499 THEN 'Type 2'
        when registrations < 500 and annualfee > 500 THEN 'Type 3'
        when registrations >= 500 and annualfee < 499 THEN 'Type 4'
    else
        'hasNulls'
    END as regSize
    , *
from
    cls.cars
limit 1000;


regsize      year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
Type 3       2008  Ida           Yes             Bus           Bus                [...]
Type 3       2011  Jasper        Yes             Moped         Moped              [...]
Type 1       2012  Harrison      Yes             Truck         Truck            3 [...]
Type 3       2015  Palo Alto     No              Trailer       Travel Trailer     [...]
Type 1       2016  Adair         Yes             Truck         Truck            3 [...]
[...]
```

In the query above if there is a Null `registration` or `annualfee` then that row will fail the Boolean clauses on part of the `CASE` statement, resulting in those rows being caught in the `ELSE` condition.

- Note that you can use a `CASE` statement in a `WHERE` clause, though it uncommon to do so. What does the following do?

```
select * from cls.cars
where
    case
        when registrations < 100 then 1
        when registrations between 200 and 300 then 2
        when registrations > 500 then 3 end = 2;


  year  countyname    motorvehicle    vehiclecat    vehicletype      tonnage      [...]
------  ------------  --------------  ------------  ---------------  ---------  - [...]
  2016  Van Buren     Yes             Truck         Truck            4 Tons       [...]
  2009  Lucas         Yes             Truck         Truck            4 Tons       [...]
  2015  Keokuk        Yes             Truck         Truck            4 Tons       [...]
  2008  Decatur       No              Trailer       Regular Trailer               [...]
  2009  Lee           Yes             Truck         Truck            5 Tons       [...]
[...]
```

- There is a second syntax for the `CASE` statement, which is not used as frequently. This second syntax can only handle equality constraints against a single column. An example of this syntax can be shown below where we use it to create a new columns which adjusts the annual fee paid by inflation.

```
select
    case year
        WHEN 2005 THEN annualfee * 1.053
        WHEN 2006 THEN annualfee * 1.051
        WHEN 2007 THEN annualfee * 1.05
        WHEN 2008 THEN annualfee * 1.04
        WHEN 2009 THEN annualfee * 1.038
        WHEN 2010 THEN annualfee * 1.035
        WHEN 2011 THEN annualfee * 1.03
        WHEN 2012 THEN annualfee * 1.01
        WHEN 2013 THEN annualfee
    end as annualfeeInflation
from cls.cars;


  annualfeeinflation
--------------------
             707.2
            1427.58
          312951

[...]
```

When using this syntax we first specify which column we are going to compare on (in this case that column is year). For each row the year column is compared against the value after `WHEN` and, if that conditional is true, the `THEN` clause is evaluated.

- A useful application of the `CASE` statement is dealing with divide by zero. Previously we had dealt with division by zero problems by removing those rows using a `WHERE` clause. If, instead of removing that row, we wish to keep it but return a different value we can use `CASE`:

```
select
    case
        when annualfee > 0 then registrations / annualfee
        else null
    end as regPerDollar
from
    cls.cars;


  regperdollar
--------------
    0.00735294
    0.142857
    0.0162013
    0.0198773
    0.0187523
[...]
```

The `annualfee` values which are either Null or equal to zero will be caught by the case statement

and, rather than returning an error, the database will return a Null.

- We can use the CASE statement to implement the LEAST and GREATEST operator on two columns, but will need to be careful about nulls. Consider the following example:

```
select
    case when X >= Y then X else Y end as larg
from
    tablename;
```

In this case, if X is Null, then Y is returned. However, if Y is Null *the Null is returned*, which is NOT what we want. In to implement GREATEST (or LEAST) via a CASE statement we have to verify that the variable is not Null, as the query below demonstrates:

```
select
    case when Y is null or X >= Y then X else Y end as lrg
from
    tablename;
```

In this case if Y is Null then X is returned, no matter the value in X while if X is Null then Y is returned, no matter Y's value.

4 The DISTINCT Operator

- The DISTINCT operator can be used in a number of ways in SQL. The first way that we will describe is how it can be used is to remove duplicates from the data that is being returned.
- If we want to know what years are in the Iowa cars table we can run the following command:

```
SELECT DISTINCT year from cls.cars;


  year
------
  2013
  2021
  2015
  2008
  2010
[...]
```

which is a list of every distinct year in the table. We can combine this with the order by command to see an ordered list of the years in the database:

DRAFT

```
SELECT DISTINCT
     year
FROM
    cls.cars
ORDER BY year;


  year
------
  2005
  2006
  2007
  2008
  2009
[...]
```

- When learning SQL, it helps to think of `SELECT` and `SELECT DISTINCT` as two different functions. `DISTINCT` is not modifying a column, it is more fundamentally changing what is returned.
- `DISTINCT` is computationally expensive. Novice query writers often make the mistake of putting it in queries when it is not required and causing the queries to be slower than necessary.
- Let us use the following dataset to understand how Nulls and multiple columns are handled. The table "BillPaid" contains information from a credit card company. In particular, it contains information about if a person paid their bill at the end of each month. The column `paytype` represents how the Person paid their bill and is Null if a person did not pay. If a person didn't pay, the amount is zero to zero.

PersonID	Month	Paid	paytype	Amt
1	1	1	Visa	15
1	2	1	Visa	100
1	3	1	Visa	15
2	1	1	Visa	25
2	2	0	NULL	0
2	3	1	Visa	25
3	1	1	Check	10
3	2	0	NULL	0
3	3	0	NULL	0

Figure 3.1: "BillPaid" Table

- As before, we can use `DISTINCT` on a single column:

DRAFT

```
select distinct PersonID from cls.BillPaid;


  personid
----------
         3
         2
         1
```

as well as on multiple columns:

```
select distinct PersonID, PayType from cls.BillPaid;


  personid  paytype
----------  ---------
         3
         3  Check
         2
         2  Visa
         1  Visa
```

Note that this command does not create any data – only takes the unique entries by row. Also demonstrated is that Null is handled as if it was its own, unique, value.

- A common error with `DISTINCT` is trying to sort on a column which is not in the `SELECT`. Consider the following query:

```
select distinct PersonID from cls.BillPaid order by amt desc;
```

Looking at the table, we can see that PersonID #1 has a value equal to 100, which is larger than any other value – so should it go first? At the same time, PersonID #2 has a value of 25, which is larger than PersonID #1 in months 1 and 3, so should it be first? Since the database is not sure which to do, it does something different: it responds with an error.

```
ERROR:  for SELECT DISTINCT, ORDER BY expressions must appear in select list
```

- Importantly, `DISTINCT` and `ORDER BY` *can* be used at the same time, but only if the column being sorted is the same one as the column being made distinct, as can be seen in the query below.

```
SELECT distinct amt from cls.BillPaid order by amt desc;


  amt
-----
  100
   25
   15
   10
    0
```

DRAFT

5 Subqueries (IN, ANY, ALL)

- Up to this point, we have used SELECT and simple WHERE clauses to choose which rows and columns to return in a query. Simple WHERE clauses allow us to choose rows based on other data within that row, but not on information outside that row. In this section we will write subqueries to filter rows based on data not present in that row. We will continue to use Table 3.1, the "Bill Paid" table.

  Looking over this table, you can see that there are three people who had bills. To write a query which identifies missing payments we could write the following query:

```
select
    *
from
    cls.BillPaid
where
    Paid = 0;


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        2       0                 0
         3        2       0                 0
         3        3       0                 0
```

  Which will return three rows, two from person #3 and one from #2.

- Assume we want to analyze *all the rows* from people who have ever missed a payment. The WHERE clause above will not work in this scenario since we need to know information about rows outside the one being evaluated. In this case we use the IN clause and a subquery:

```
select
    *
from
    cls.BillPaid
where
    personid IN (select personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

  The IN clause used with the WHERE is evaluated exactly as you would expect: for each row in the table, the query determines if that countyname is in the list generated by the subquery. These types of subqueries are called *uncorrelated* because nothing in the subquery references anything outside that subquery.

- When using this syntax, the subquery needs to return a single column of data. Looking at the above we can see that the subquery above satisfies this constraint.
- The opposite of `IN` is `NOT IN`, which only accepts rows do not match the contents of the subquery. For example, the following would return only the rows associated with people who have never missed a payment:

```
select
    *
from
    cls.BillPaid
where
    personid NOT IN (select personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
```

- Note that the subquery syntax does not look at the name of the column within the subquery. For example, the following query will work as well:

```
select
    *
from
    cls.BillPaid
where
    personid IN (select personid as sillyColumnName
            from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

- Keep in mind that the reason we need to use this syntax is because we need information that is outside of the current row to evaluate the current row. A simple `WHERE` clause can only access the information in the current row.
- The `IN` clause can be used without a `SELECT` as a subquery:

```
select
    *
from
    cls.billpaid
where
    personid in (1,2);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
         2        1       1  Visa          25
         2        2       0                 0
[...]
```

In this case there is no official subquery – the query itself contains the data to be filtered on.

- An important consideration when writing subqueries is the use of `DISTINCT` in the subquery itself. The `IN` operator verifies if a particular value is within a list. If the list has duplicates then the verification process will take longer. In the above example, the subquery returns 3 values (2,3,3), two of which are duplicates. When making the comparison, having duplicates in the subquery list will result in an inefficient comparison. To avoid this, we generally add a `DISTINCT` operator to the subquery:

```
SELECT
    *
FROM
    cls.BillPaid
WHERE
    personid IN (select distinct personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

This will yield a more efficient query. Because the dataset is small, the difference in this query will be negligible, for larger datasets this change may be necessary for the query to run in a manageable amount of time.

- There are two other operators that are used in a similar fashion, though I do not find myself using them frequently, `ANY` and `ALL`, which are used in the following manner:

```
WHERE column [OPERATOR] ANY/ALL (SUBQUERY)
```

AFT

- For example, consider the following two examples:

```
SELECT *
FROM cls.BillPaid
where amt
    <= ALL (select amt from cls.billpaid where personID = 1);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        3       1  Visa          15
         2        2       0                 0
         3        1       1  Check         10
         3        2       0                 0
[...]
```

```
SELECT *
FROM cls.BillPaid
where amt
    <= ANY (select amt from cls.billpaid where personID = 1);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
         2        1       1  Visa          25
         2        2       0                 0
[...]
```

In the first example, only those rows where `amt` is less than all values from PersonID #1 (15,100,15), are returned. This would return the 4 rows with amt = 0 and amt =10, this is equivalent to ≤ 15. The second query, on the other hand, only checks to see if it less than a single value within that list, so this is equivalent to ≤ 100, which returns all rows in the table.

6 Correlated Subqueries

- A correlated subquery references the outer query within the subquery. For example, consider the following query:

```
select
    *
from
    cls.cars as A
where
    vehicletype = 'Motorcycle'
    and year <> 2010
    and countyname in
    (select
        countyname
    from
        cls.cars as B
    where
        A.countyname = B.countyname
        and B.year = 2010
        and B.vehicletype = 'Motorcycle'
        and A.registrations > B.registrations);


  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2016  Jasper        Yes             Motorcycle    Motorcycle                    [...]
  2011  Ringgold      Yes             Motorcycle    Motorcycle                    [...]
  2013  Clayton       Yes             Motorcycle    Motorcycle                    [...]
  2013  Grundy        Yes             Motorcycle    Motorcycle                    [...]
  2016  Davis         Yes             Motorcycle    Motorcycle                    [...]
[...]
```

This will return all motorcycle rows, for each county that have more registrations than that same county's registrations for 2010. For example, Lucas county has the following number of registrations for each year, for motorcycles:

```
select
    year
    , registrations
from
    cls.cars
where
    countyname ='Lucas'
    and vehicletype = 'Motorcycle'
order by 1;


  year    registrations
------  ---------------
  2005              530
  2006              586
  2007              606
  2008              592
  2009              587
[...]
```

This statement will only evaluate positive in 2006 and 2007, the rows that have more registrations than 2010. To furuther understand this query, think through each row as an item within a loop, with the subquery being evaluated each time.

In Lucas, year 2005, for example, the subquery will look like :

```
(select
     countyname
from
    cls.cars as B
where
    'Lucas' = B.countyname
    and B.year = 2010
    and B.vehicletype = 'Motorcycle'
    and 530 > B.registrations);


countyname
------------
```

This subquery will return Null since no `countyname` will match the constraints in the where clause. Since it returns Null, the outer where clause evaluates False and 2005 is not returned.

- If we wanted to find all counties which increased the number of motorcycle registrations from 2005 to 2006 we could write the following query:

```
select
    countyname
from
    cls.cars as A
where
    A.year = 2006
    and A.vehicletype = 'Motorcycle'
    and countyname in
        (select countyname
            from
                cls.cars as B
            where
                year = 2005
                and A.countyname = B.countyname
                and B.vehicletype = 'Motorcycle'
                and A.registrations > B.registrations);


countyname
------------
Adair
Osceola
Madison
Worth
Hancock
[...]
```

- Correlated subqueries are costly computationally since the subquery is reevaluated for row, you can think of them as FOR LOOPS in SQL. They are also incredibly difficult to read. **Generally**

speaking, they should be avoided. We will learn techniques for avoiding them later.

- There is one interesting case for correlated subqueries, which is identifying the "first row" of a particular group. Consider the following query:

```
select
    a.countyname, a.registrations
from
    cls.cars as a
where
    a.registrations =
        (select
            registrations
        from
            cls.cars as b
        where
            a.countyname = b.countyname
        order by
            b.registrations desc
            limit 1)
```

Note that this query will take an incredibly long time to evaluate.[1] It will return, for each county, the *largest* number of registrations for a row. In other words, correlated subqueries can be used to determine the first value for a particular row. This same technique can be used to determine the maximum or minimum value of a particular column within subgroups. Later on we will learn much smarter techniques for doing this.

DRAFT

[1] I stopped it after one minute so I'm not sure how long it takes in total.

DRAFT

Chapter 4

Database Internals: Transactions

DRAFT

Contents

1 REDO / COMBINE NEXT SECTIONS **65**
2 Table Creation and Deletion **65**
3 Database Operations: CRUD **65**
4 Creating Tables, Constraints and Deleting tables **66**
5 Altering Tables **68**
6 Inserting, Copying, Updating and Deleting **68**
7 Transactions and ACID **69**
8 Isolation Levels in Relational Databases **73**
9 Why do we care (NoSQL)? **78**
10 NoSQL **80**
11 Transaction Implementations [TBD] **81**

DRAFT

1 REDO / COMBINE NEXT SECTIONS

2 Table Creation and Deletion

- Up to this point we have glossed over the details of how tables and schemas are created. There are a few reasons for doing this:
  1. In professional settings it is relatively uncommon to be creating tables. By definition they are created only once – as opposed to accessing them which can be done much more frequently.
  2. The considerations for how a table is created are very specific to the database variant. Data types are treated differently within different database.
     - For example, in Postgres, there is no "fixed length string type". Even if you define a column as a CHAR, the database treats it as a VARCHAR. Other SQL variants do not follow this same pattern and therefore the amount of space taken up by each column and its relative efficiencies are different in different variants.
     - These types of physical layer differences can mean that optimizations and best practices that work on one database variant may not work on another.
  3. On top of the the physical layer differences, variants also express different syntax in their creation statements.
     - Just like the rest of SQL, the core syntax displays is similar between variants. However, once you step outside the basics of creating a table, the syntactical differences start increasing.
  4. Last, but not least, questions regarding creating a table rarely appear in data science or data analyst interviews.
- Given the above we won't spend too much time understanding the syntax of creating a table, though we will cover the important aspects involved.
- The basic syntax for creating a table follows the following form:

```
CREATE TABLE schema.table_name
(
        column_1_name data_type,
        column_2_name data_type,
        column_3_name data_type
);
```

  It is a pretty simple syntax overall – name and datatypes are the only two required arguments.

3 Database Operations: CRUD

There are four essential operations for any database which we abbreviate **CRUD**:

- **C**reate: New data can be added to a database[1]
  - CREATE: Create a table for data to be stored.
  - INSERT: Put data into a table.

[1]Relational databases break up the process of data creation into two steps. The first is to create a container for the data (via CREATE) and then second is to populate that table, using either a COPY or INSERT command.

DRAFT

  - COPY: Bulk data loading operation.
- **R**ead: Retrieve previously stored data.
  - SELECT: Returns data to the client.
- **U**pdate: Previously stored data can be changed in a database.
  - UPDATE: Changes already stored data.
  - ALTER: Changes the structure of a table.
- **D**elete: Previously stored data can be removed
  - DROP: Removes a database object
  - DELETE: Removes data from a table
- Many databases, including newer versions of PostgreSQL include a command UPSERT, which is a portmanteau of UPDATE and INSERT. UPSERT will insert a new row, unless that row is already present, in which case it will update.
- This breakdown can be found in a variety of settings, including when used in RESTful APIs, which is the basic architecture used for client server communication via the internet. As can be seen in Table 4.1, each operation above maps directly to a type of HTTP request.

Operation	HTTP Method
Create	PUT
Read	GET
Update	PUT
Delete	DELETE

Table 4.1: CRUD map to HTTP Requests

In this course we aren't as interested in operations outside of SELECT. For most data analysts and data scientists, 99.999% of the queries that they will write are retrieving data.

4 Creating Tables, Constraints and Deleting tables

- To create an empty table we use the `CREATE TABLE` command and specify the table name and the columns therein.
- At the time that the table is created we specify the entire table schema and specifically the column names and the data types of those columns.
- If we were create a table which contains the names of people and the balances of their bank accounts, we would use the following command:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);
```

which would create a table with two columns, the first being called aName and the second aBal. The first column is a `varchar` – or variable length character string and the second is an integer.

DRAFT

- This command also allows us to specify additional properties of the table and columns within the table.
- There are a few classes of properties that we can specify:
  1. **Storage information:** Specific configuration regarding how the data is stored at the physical layer (beyond the scope of this course)
  2. **Constraints:** Rules that the data within the table must abide by *before* the data is inserted into the table. We will talk a bit about this below.
  3. **Index information:** What indexing strategies should be used in the table (we'll talk about this later).
  4. **Triggers:** A "trigger" is a piece of procedural (usually non-SQL code) that is run when a particular SQL command is run. For example, you could create a trigger which would send an email or raise an alert if a particular type of SQL command is run on the table. This is beyond the scope of this course.
- Constraints and Triggers are powerful tools within a database because they allow us to verify data as it is being loaded and execute additional commands upon the data being updated or changed. The downside of both of these powerful tools is that they put additional strain at on the database and therefore in certain situations they are avoided. For example, databases with very large write-loads (e.g. they are writing a lot of data very quickly) will usually avoid adding constraints because the cost of checking those constraints can bottleneck the system.
- The two most frequently used constraints are `NOT NULL` and `UNIQUE` which enforce the obvious on a column (or set of columns). For example, we could write the following as our create table:

```
CREATE TABLE cls.balances2 (
        aName varchar(30) UNIQUE NOT NULL
        , aBal int NOT NULL
);
```

  which would prevent the database from allowing any data to be inserted with either a Null abalance or a Null or Non-unique aName.
- Uniqueness constraints can also be created at the table level on groups of columns. It is possible to encounter a situation where we wanted a pair (or more) of columns to be unique together we could also specify this.
- There are tons of other constraints, but they are not as common and their syntax is frequently SQL variant specific. Here is a fun constraint on aName:

```
CREATE TABLE cls.balances3 (
        aName varchar(30) UNIQUE NOT NULL CHECK (trim(aname) <> '' AND length(aname) > 5 )
        , aBal int NOT NULL
);
```

  This would check to make sure that aName is (1) unique, (2) not null, (3) not equal to an empty string and (4) length larger than 5.
- To delete a table from the database we use the command `DROP`. To drop the tables we have created above you could use the following commands:

DRAFT

```
DROP TABLES cls.balances2;
DROP TABLES cls.balances3;
```

- A useful argument for these two commands specifies behavior in the case that the table already or does not exists. The arguments `IF NOT EXISTS` and `IF EXISTS` which are used on `CREATE` and `DROP` respectively and will only create if the table does not exist and drop if it does. These are useful when loading and dropping data to avoid errors being returned. You can find examples of these commands in the sql-data repository and specifically the functions contained in the `load_data.py` file.

5 Altering Tables

- To change the structure of a table we use the command `ALTER`.
- The most common operations that we do with `ALTER` are adding, dropping and modifying columns.
- These operations should not be undertaken lightly as they can be incredibly expensive for the database. Mentally, you should equate running one of these commands with reading and rewriting the entire table.
- Here are two examples:

```
ALTER TABLE cls.balances ADD COLUMN new_col_1 varchar(10);
ALTER TABLE cls.balances DROP COLUMN new_col_1;
```

6 Inserting, Copying, Updating and Deleting

- We use the `INSERT` command to put small amounts of data into a table. For larger amounts we use the `COPY` command.
- While the `INSERT` command facilitates getting data into a table a few different ways, we'll focus on two: providing values directly and loading values generated from a query.
- Consider the following commands, which create a table and then insert values into it from the output of another query.

```
create table cls.unique_county_names (
countyname varchar(20) NOT NULL UNIQUE
);

insert into cls.unique_county_names
        (select distinct countyname from cls.cars);
```

- The other common way to use `INSERT` to put data into a table is by providing values, such as in the example below:

DRAFT

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);
```

- Each row is put in parenthesis in this method.
- Inserting data into a table is (unsurprisingly) costly and should not be undertaken lightly.
- If we wish to change data within a table, while maintaining the same data types we use the `UPDATE` command, as in the following example:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);

UPDATE cls.balances set aName = 'Nicholas' where aName = 'Nick';
UPDATE cls.balances set aBal = 0;
```

- To remove data from a table we use `DELETE`:

```
DELETE FROM cls.balances where aName = 'Nicholas';
```

- If we want to drop all rows from a table we use `TRUNCATE`, which is a high performance delete when removing *all* rows.

```
TRUNCATE TABLE cls.balances;
```

DRAFT

7 Transactions and ACID

- Consider the following table and queries:

```
CREATE TABLE cls.balances (
        aName varchar(30)
        , aBal int
);

INSERT INTO CLS.balances VALUES ('Nick', 1000), ('Jim', 300)
        , ('Judy', 100), ('Jean', 1500);

--Assume that Nick makes a deposit of 100:
update cls.balances set aBal = aBal + 100 where aName = 'Nick';

--Assume that Jim withdraws 200:
update cls.balances set aBal = aBal - 200 where aName = 'Jim';

--Assume that Jean drops her account:
delete from cls.balances where aName = 'Jean';

--Assume that Julian creates an account with 250:
insert into cls.balances values ('Julian', 25);
```

The resulting table would have four rows:

```
select * from balances;

 aname  | abal
--------+------
 Judy   |  100
 Nick   | 1100
 Jim    |  100
 Julian |  250
 (4 rows)
```

- A **transaction** is a unit of work which is to be treated as a single entity within a database. A transaction may consist of *multiple* queries.
- Transactions are important in a few specific instances: (1) If there are multiple users/sessions accessing and manipulating the database and (2) If queries are required be executed together.
- If nothing within the transaction is changing the state of the database (think the underlying data), then transactions become much less important.
- If there is only a single user interacting with the database in a single session, then the implications of transactions aren't that important, but this is not usually the case.
- The lifecycle, or transaction process looks like the following:

  1. Begin the transaction
  2. Attempt all statements within the transactions:
  3. If they are successful then *commit* the transaction.
  4. Otherwise (at least one statement fails): then *rollback* the transaction.

5. End the transaction.

- The phrase *commit* is technical – it's when whatever changes to the database are set so that other users see them.
- If a commit fails, the database will *rollback* to its previous state.
- Consider the following example, based on the *balances* table from the previous section, where Nick gives $100 to Jim.

```
BEGIN;

UPDATE CLS.BALANCES SET ABAL = ABAL + 100 where aname = 'Jim';

UPDATE CLS.BALANCES SET ABAL = ABAL - 100 where aname = 'Nick';

COMMIT;
```

- In this example the two commands regarding Nick and Jim are treated as a *single* command. We want this to be the case – if something goes wrong, the state of the database returns to the state that it was at the start of the transaction.
- Most SQL clients have an *AUTOCOMMIT* feature, which runs every query as its own transaction. Some SQL clients will use the semi-colon as a transaction divider. For most of the SQL clients I've seen and used these parameters are configurable.

```
BEGIN;

UPDATE CLS.BALANCES SET ABAL = ABAL + 100 where aname = 'Jim';

--Query below has an error and thus the database will be returned to the
--state before the BEGIN
UPdt CLS.BALANCES SET ABAL = ABAL - 100 where aname = 'Nick';

COMMIT;
```

- Consider the queries in Figure 4.1. Each column represents a connection to the database while time increases downward. Any query not in a transaction block is its own transaction.
- Importantly, the database keeps the transaction behavior separate between the connections until those transactions are committed.
- What makes a good transaction? Transactions in Relational Databases have the following properties, generally referred to as ACID:
  - **A**tomic: Transactions are treated as a single unit. If there are 9 queries within a transaction and the 8th one fails, the seven queries that preceded it are not committed to the database.
  - **C**onsistent: A transaction is processed if and only if it does not violate any system rules. For example, if you try to put a string in an integer field then the transaction will fail.
  - **I**solation: Transactions that are executed concurrently behave *appropriately.* More on this in the next section.
  - **D**urable: Once a transaction is committed, it is committed perpetually. If the system reboots or restarts after a transaction is committed then you can expect that transaction to have gone

Figure 4.1: Transaction Demonstration

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
200
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)
```

through. After a restart the system will not contain the previous state of the database.

8 Isolation Levels in Relational Databases

- The Isolation aspect of ACID is complex. Specifically, Isolation refers to how the database operates in the case when multiple transactions conflict with each other. For example, if two transactions try to update the same row of data within a table then isolation describes how this conflict gets resolved.
- When I think of Isolation I don't think of it as a "single property" – like Atomicity or Durability, but more as a spectrum of behaviors that a database can demonstrate in relation to specific conflicts.
- The ANSI SQL standard specifics four different levels of transaction isolation. These isolation levels are defined by what phenomena they disallow, as can be seen in Table 4.2. Note that these Isolation levels are listed from "weakest" to "strongest". We will go over each below.[2]

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed[3]	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed[4]	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Table 4.2: ANSI standard for Isolations

1. **Dirty Read:** A dirty read occurs when a user sees data that is not committed. Consider the following example, once again starting from the transactions in 4.2 (which are similar to those in Figure 4.1 above). A dirty read has occurred because, in connection #2 the balance shows the uncommitted 300 which is a contamination between transactions.

   We can't give a proper demonstration of a dirty read in Postgres because, even at the lowest Isolation Level ("Read Uncommitted") Postgres's Isolation system won't allow it.

2. **Nonrepeatable Read:** A nonrepeatable read occurs when the state of database has changed between two reads within a transaction. Specifically, nonrepeatable reads occur when the *value within a row* has changed between two `SELECT` statements. Consider the set of transactions in Figure 4.3 which demonstrate this occurring as the second connection returns two different values for the same query *within* a transaction.

3. **Phantom Read:** A Phantom Read occurs when the state of database has changed between two reads within a transaction and it *effects which rows are returned.* Consider the example in Figure 4.4. In this example the rows that are being returned are incorrect, in that the second transaction shouldn't see the changes from the first transaction as they have not been committed.

4. **Serialization Anomaly:** A serialization anomaly occurs when the result of concurrent transactions is different depending on the order. For example, consider the transactions in Figure 4.5. In this case, the order of the transactions, which are concurrent, determines the final table values.

[2]There are a number of people who have strong opinions about the effectiveness of these designations. *A Critique of ANSI SQL Isolation Levels* published by Microsoft Research's Advanced Technology division provides a good starting point for this discussion.

DRAFT

Figure 4.2: Dirty Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname   | abal
--------+------
Judy    |  100
Julian  |   25
Jim     |  200
Nick    | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;
aname   | abal
--------+------
Judy    |  100
Julian  |   25
Jim     |  200
Nick    | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)
```

Figure 4.3: Nonrepeatable Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = ABAL + 100
        WHERE aname = 'Jim';

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

UPDATE CLS.BALANCES
        SET ABAL = ABAL - 100
        WHERE aname = 'Nick';

COMMIT;
```

Connection #2

```
BEGIN;

SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
200
(1 row)

SELECT ABAL from CLS.BALANCES
WHERE aname = 'Jim';
abal
------
300
(1 row)

COMMIT;
```

Figure 4.4: Phantom Read Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 50
        WHERE aname = 'Jim';








COMMIT;
```

Connection #2

```
BEGIN;
SELECT * FROM CLS.BALANCES;
aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)








SELECT ANAME from CLS.BALANCES
WHERE ABAL <= 100;
 aname
--------
 Julian
 Jim
(2 rows)




COMMIT;
```

Figure 4.5: Serialization Anomaly Example

Connection #1

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 50
        WHERE abal >= 100;

COMMIT;
```

Connection #2

```
SELECT * FROM CLS.BALANCES;

aname  | abal
--------+------
Judy   |  100
Julian |   25
Jim    |  200
Nick   | 1000
(4 rows)

BEGIN;

UPDATE CLS.BALANCES
        SET ABAL = 25
        WHERE abal >= 100;

COMMIT;
```

DRAFT

- Referring back to Table 4.2, there are multiple isolation levels defined which either allow or disallow the above cases to occur. In Postgres, we can see the current isolation level as well as set the isolation level within a transaction using the commands:

```
BEGIN;

sql_class=*# show transaction isolation level;
 transaction_isolation
-----------------------
 read committed
(1 row)

set transaction isolation level Serializable;
SET

show transaction isolation level;
 transaction_isolation
-----------------------
 serializable
(1 row)

COMMIT;
```

- By switching the isolation level associated with the transaction we can change the databases behavior in these situations.
- What happens if we trigger this behavior? In PostgreSQL, the offending transaction will either "lock" or "fail" due to an error. In the case that the isolation level is set to `serializable` and two conflicting transactions occur, the database will return an error like:

```
ERROR:  could not serialize access
        due to read/write dependencies
        among transactions
DETAIL:  Cancelled on identification
         as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

- A "lock" on the other hand occurs when the database realizes that two processes are vying for a single resource and chooses to "block" one of those processes until something gets resolved. In the case of a lock occurring user intervention maybe required if certain time out thresholds aren't met.
- Why does this matter?

9 Why do we care (NoSQL)?

- THIS IS STILL UNDER CONSTRUCTION.
- Why is ACID important to think about? It seems that these properties seem "obvious", in the sense that they are things that we want in a database. While that maybe true, making sure that a database is ACID compliant is costly in a few ways.
- While there is no "official" definition of NoSQL, one way that other database systems distinguish

DRAFT

themselves is by not being fully ACID compliant. Amazon's Cloud Service FAQ notes:[5]

> NoSQL databases often trade some ACID properties of traditional relational database management systems (RDBMS) for a more flexible data model that scales horizontally. These characteristics make NoSQL databases an excellent choice in situations where traditional RDBMS encounter architectural challenges to overcome some combination of performance bottlenecks, scalability, operational complexity, and increasing administration and support costs.

- Is this the *only* thing that distinguishes NoSQL databases from traditional RDMS systems? Absolutely not.
- There are tons of very reasonable ways that a database can relax the rules around an RDMS and be called a NoSQL database – relaxing the transaction assumptions is just one of the ways that it can be done.
- Another common distinction between RDMS and NoSQL databases is how information within the database is stored and it's ability (or inability) to be used internally to the database, such as in joins.
- MongoDB ("Mongo") presents an interesting case study in what it means to be a NoSQL database. When Mongo was originally released, the first versions had known limitations around their durability. A (relatively) at the time, database meme was that MongoDB was "marketing pretending to be a database" or that it was the SnapChat of databases (see the image in 4.6 for something that was passed around in the early 2010's).

Figure 4.6: Something to keep in mind! Source unknown.

[5] https://aws.amazon.com/nosql/?nc1=f_cc

The original versions of MongoDB were well known to not be durable

MongoDB, for example, is not as atomic as other databases. In Mongo for example, a WHERE clause may not return a matching row if the row is in the middle of an update, even if both pre- and post-update that row would match the WHERE clause. Amazon's Redshift system also relaxes some consistency measures to make sure that data can load quickly.

10 NoSQL

Types of databases and NoSQL.

Generally NoSQL relaxes some of the constraints that can be found in relational databases.

DRAFT

- **Key-Value Stores**: Key-value stores are databases that are organized around two columns, a key and a value. Values in this case generally have set, fixed schemas. Examples of this type of database include HBase, Amazon's Dynamo DB and Cassandra. The common use case of this data is severing user-specific data, such as saved game files or account information. In most of these cases there is enough structure on the data in the value the database itself can use that information to it's advantage.
- **Document Stores**: This type of database structure is similar to the Key-Value store, expect that instead of the record being a value, the record is a document, which is different from a value in that it has a less fixed size schema. The reason that this is called a document store, as opposed to a key value store is to make it clear that the database isn't going to try to use information regarding the values in optimizing its data structure. Examples of this include MongoDB and common use cases include storing network information.
- **Relational databases**: Relational databases are the most common type of database. The key feature of these databases is that the information within the database is expected to be stored in multiple tables that join together. In terms of performance, the database engines expect to use joins and provide structure on optimizing tables along different dimensions to facilitate multiple joins. There are two major classes of these databases and the properties of each are sufficiently different that we cover them separately.
  - **Row-based databases**: A Row-based database, such as PostgresSQL and Postgres store data in rows on the hard drive. In other words, if you were looking at a row-column entry on the hard drive and then read the next big of data, it would contain the next column in the database, alone that same row. This means that loading whole rows, once a row has been identified, is performant. If you were working at a bank-terminal and were looking up customer information, this would make sense, since once the customer's location has been identified on the hard drive, all the ancillary information about that customer is right there. The first two databases that we are going to talk about in this class, PostgresSQL and Postgres are both row-based relational databases.
  - **Columnar databases**: Column-based ("Columnar") databases, store information in columns on the hard drive. This type of database technology is relatively recent, examples include Vertica, SAS HANA and Amazon's redshift. These databases exist to assist data analysts, as the most common operations that data analysts do are not row, but column based. For example, in a row-based database, finding the average of a column is difficult because the information will be scattered over the entire data partition. Columnar databases are optimized for this type of operation by placing the data in column order. The final database that we are going to consider in this class, Amazon Redshift, is a columnar database.

11 Transaction Implementations [TBD]

DRAFT

DRAFT

Chapter 5

Aggregations

DRAFT

Contents

1 Introduction to MTA data set **85**

2 GROUP BY clause **86**

3 Column numbering syntax **91**

4 Aggregates and CASE Statements **93**

5 Named Subqueries **95**

DRAFT

1 Introduction to MTA data set

- In this section we are going to introduce another data set, the NY MTA dataset, which contains information on the number of cars that pass certain plazas between January 1st, 2010 and January 7th, 2017, about 7 years of data.
- Looking at the dataset, we see that it is a long (or tall) dataset, with 6 columns. The data represents the number of cars that go through different toll plazas in the city by hour. The data is divided between cars which paid via EZ-pass and cash and split between those drivers heading away from the city ("O" direction) and those heading into the city ("I").

```
select * from cls.mta limit 10;


  plaza  mtadt         hr  direction      vehiclesez    vehiclescash
-------  ----------  ----  -----------  ------------  --------------
      2  2013-10-14    16  I                    2469             336
      2  2013-10-14    16  O                    2393             473
      2  2013-10-14    17  I                    2853             425
      2  2013-10-14    17  O                    2116             417
      2  2013-10-14    18  I                    2575             394
[...]
```

- For example, if we want to see the number of cards which are heading outbound between 2 and 3 am on the 15th of June, 2015 over the Robert F. Kennedy Bridge Manhattan Plaza (Triborough bridge into Manhattan, which is Plaza #2), can be found by writing the following query:

```
select
    *
from
    cls.mta
where
    plaza = 2
    and direction = 'I'
    and hr = 2
    and mtadt = '2015-06-15';


  plaza  mtadt         hr  direction      vehiclesez    vehiclescash
-------  ----------  ----  -----------  ------------  --------------
      2  2015-06-15     2  I                     173              58
```

- A unique row in this dataset is denoted by the items in the WHERE clause – plaza, mtadt, hr and direction.
- If we wanted look at the total number of cars, for each hour, that go through Plaza #2 we could do the following:

```
select
    plaza, mtadt, hr
    , vehiclesez + vehiclescash as totalCars
from
    cls.mta
where
    plaza = 2
    and direction = 'I'
order by mtadt, hr;


  plaza  mtadt         hr    totalcars
-------  ----------  ----  -----------
      2  2010-01-01     0          747
      2  2010-01-01     1          903
      2  2010-01-01     2          742
      2  2010-01-01     3          501
      2  2010-01-01     4          456
[...]
```

2 GROUP BY clause

- Up until this point we have been slicing data, removing rows and columns. The next syntax we will study aggregates, or collapses, data into a smaller number of rows. In other words, this operation now looks between rows in order to undertake its calculation. Importantly, this operation defines subsegments of the table that are treated as a single group.
- Consider the following query:

```
select
     MAX( vehiclescash) as maxcash
    , plaza
from
    cls.mta
group by plaza;


  maxcash    plaza
---------  -------
     1352        1
     1040        2
     1594        3
     1368        4
      674        5
[...]
```

  GROUP BY to combines similar values. This query combines data by plaza and returns the maximum number of cars that pay cash in any hour through that plaza.

- This query will return 10 rows, one for each plaza. The query calculates the the maximum value of

DRAFT

`vehiclescash` for by `plaza`.

- The GROUP BY clause is applied and written *after* the WHERE clause. If a WHERE clause removes a row then that row will not be aggregated via the function.

```
select
     MAX( vehiclescash) as maxcash
    , plaza
from
    cls.mta
where
    plaza = 2
group by plaza;


  maxcash    plaza
---------  -------
     1040        2
```

- GROUP BY requires every column within the SELECT clause to be either inside a function or part of the GROUP BY. The following query yields an error:

```
select
     MAX( vehiclescash) as maxcash
     , hr
    , plaza
from
    cls.mta
group by plaza;

ERROR: column "cls.hr" must appear in the GROUP BY
clause or be used in an aggregate function
```

- Other aggregate functions include average ("AVG"), minimum ("MIN"), count ("COUNT") and sum ("SUM"):

DRAFT

```
select
    plaza
    , min( vehiclescash) as minveh
    , count( vehiclescash) as ctveh
    , sum( vehiclescash) as sumveh
    , avg( vehiclescash) as avgveh
from
    cls.mta
group by plaza
order by avg(vehiclescash) desc;


  plaza    minveh    ctveh    sumveh    avgveh
-------  --------  -------  --------  --------
     11         0    61488  38181458   620.958
      3         0   122976  67000523   544.826
      1         0   122976  54359482   442.033
      9         0   122976  53530379   435.291
      2         0   122976  38009405   309.08
[...]
```

- Implicit GROUP BY: If every column within a select statement is an aggregate function then the query will still run, even if it does not have GROUP BY put down explicitly:

```
select
    sum( vehiclescash ) as sumveh
    , avg( vehiclescash ) as avgveh
from
    cls.mta;


   sumveh    avgveh
---------  --------
330901032   283.858
```

In this case, the entire table is treated as a single group within the GROUP BY.

- There is also a special aggregation: COUNT(DISTINCT XXX), which returns the number of unique values within a given group:

```
select
    count(distinct plaza) as plazact
from
    cls.mta;


  plazact
---------
       10
```

Note that COUNT DISTINCT counts the number of unique non-null entries.

- We can include multiple columns within the GROUP BY and it will calculate the functions among unique combinations of the columns selected. For example:

```
select
    plaza
    , mtadt
    , sum(vehiclescash + vehiclesez) as totalcars
from
    cls.mta
group by plaza, mtadt
order by plaza, mtadt;


  plaza  mtadt          totalcars
-------  ----------  -----------
      1  2010-01-01        57606
      1  2010-01-02        63405
      1  2010-01-03        59496
      1  2010-01-04        72610
      1  2010-01-05        72880
[...]
```

- What if I forget to include an AS?

```
select
    count(vehiclesez)
    , max(vehiclesez)
    , max(vehiclescash)
from
    cls.mta;


  count    max    max
-------  -----  -----
1165728   8345   2116
```

Without the AS, the database returns the column with the name of the aggregate function.

- Before continuing, lets answer some simple questions about the table. What percentage of cars which pass through a toll plaza during this time period use an EZ-pass?

```
select
    sum(vehiclesez)::float / (sum(vehiclesez) + sum( vehiclescash)) as pct_EZ
from
    cls.mta;


  pct_ez
--------
0.817743
```

We can see that it is around 80%.

- COUNT AND SUM can be used to return the number of rows within the table. Looking at the queries in Table 5.1 you can see that placing a number within a count returns the number of rows. Note that the second query will return the number of rows because it counts the number of '1's that appear. It is not counting the number of rows in the first column – it is counting the number of rows that would appear if every value within that column was equal to 1. Consider the following variants on this in the following table:

Syntax	What is returned
`select count(*)`	Number of rows
`select count(1)`	Number of rows
`select 2*count(*)`	Twice the number of rows
`select 2*count(2)`	Twice the number of rows
`select 2*count(-1)`	Twice the number of rows
`select 2*count(Null)`	Zero
`select 2*sum(1)`	Twice the number of rows
`select 2*sum(2)`	Four times the number of rows

Table 5.1: Examples of special syntax for counting rows

- GROUP BY treats NULL as a special, unique value. If there are NULL values in the column being grouped, they will be treated as a single group.
- Null values within aggregate functions are not straightforward. Consider the following table ("null_test") which has a two columns ("val" and "cond"), as can be seen below:

```
select * from cls.null_test;


  val  cond
-----  ------
    1  A
    2  A
    3  A
       B
```

- SUM, MAX, MIN, COUNT and AVG all ignore Null values:

DRAFT

```
select
    sum(val) as st
    , max(val) as mt
    , min(val) as mnt
    , avg(val) as at
    , count(val) as ct
    , count(distinct val) as cd
from
    cls.null_test;


  st    mt    mnt    at    ct    cd
----  ----  -----  ----  ----  ----
   6     3      1     2     3     3
```

Note that this is different then when using ORDER BY, which treats Null values as larger than any other value. Note that AVG(X) is equivalent to SUM(X) / COUNT(X). With COUNT(val), the Null is ignored. However with count(*) the Null is not ignored!

```
select count(*) as ct, count(val) as ct2 from cls.null_test;


  ct    ct2
----  -----
   4      3
```

- If the entire column is Null within a group, then each of AVG, MAX, MIN, SUM will return Null and COUNT will return zero:

```
select
    cond
    , sum(val) as st
    , max(val) as mt
    , min(val) as mnt
    , avg(val) as at
    , count(val) as ct
    , count(distinct val) as cd
from
    cls.null_test
group by cond;


cond      st    mt    mnt    at    ct    cd
------  ----  ----  -----  ----  ----  ----
A          6     3      1     2     3     3
B                                   0     0
```

3 Column numbering syntax

- As with ORDER BY we can use column numbering syntax:

```
select
    plaza
    , min( vehiclescash) as minveh
    , count( vehiclescash) as ctveh
    , sum( vehiclescash) as sumveh
    , avg( vehiclescash) as avgveh
from
    cls.mta
group by 1;


  plaza    minveh    ctveh    sumveh    avgveh
-------  --------  -------  --------  --------
      1         0   122976  54359482  442.033
      2         0   122976  38009405  309.08
      3         0   122976  67000523  544.826
      4         0   120624  21397862  177.393
      5         0   122976   7798630   63.4159
[...]
```

In the query above the number 1 in the GROUP BY clause denotes the first column in the select statement. In this case, that is "plaza"

- We can add multiple columns when using column numbering syntax. For example:

```
select
    plaza
    , mtadt
    , min( vehiclescash) as minveh
    , count( vehiclescash) as ctveh
    , sum( vehiclescash) as sumveh
    , avg( vehiclescash) as avgveh
from
    cls.mta
group by 1,2;


  plaza  mtadt         minveh    ctveh    sumveh    avgveh
-------  ----------  --------  -------  --------  --------
      1  2010-01-01       249       48     28166   586.792
      1  2010-01-02       186       48     28583   595.479
      1  2010-01-03       261       48     27272   568.167
      1  2010-01-04       143       48     26210   546.042
      1  2010-01-05       103       48     25218   525.375
[...]
```

In this query, the data is grouped by two columns: plaza and mtadt. The grouping columns are specified as "1,2".

4 Aggregates and CASE Statements

- Aggregates and CASE statements can be combined in powerful ways. Let's first count the number of rows in the database where the hour is 2 and the number of vehicles paying cash is greater than 400. As demonstrated by the query below we can use a WHERE clause to only include the rows in the table which fulfill this criteria.

```
select
    sum(1) as ct
from
    cls.mta
where
    hr = 2
    and vehiclescash > 400;


  ct
----
 256
```

- Let's say that we also wish to get the number of rows in the database where the the number of vehicles paying cash is less than or equal to 5 and the hour is 2. Because we are cutting up the data into two mutually exclusive ways we need to do something other than a WHERE clause. If we remove the rows to satisfy the first condition then we remove rows that would need to be counted in the second condition.

  We can implement both criteria using a CASE statement inside an aggregate function:

```
select
    sum( case
       when hr = 2 and vehiclescash > 400 then 1
       else 0 end) as ct1
    , sum( case
       when hr = 2 and vehiclescash < 5 then 1
       else 0 end) as ct2
from
    cls.mta;


  ct1    ct2
-----  -----
  256   1465
```

- We could also use the COUNT notation, rather than a SUM, by switching the zeros to Null:

```
select
    count( case
       when hr = 2 and vehiclescash > 400 then 1
       else Null end) as ct1
    , count( case
       when hr = 2 and vehiclescash < 5 then 1
       else Null end) as ct2
from
    cls.mta;


  ct1    ct2
-----  -----
  256   1465
```

- We can group by any column expression, including a CASE statement. In the following example we use a CASE statement to categorize different rows and then use a GROUP BY statement in order count how many of each occurs.

```
SELECT
    CASE
        WHEN vehiclescash > 400 then 'More than 400'
        WHEN vehiclescash >= 5 then 'Between 5 and 400'
        ELSE 'Less than 5'
    END as breakdown_flag
    , count(1)
    , avg( vehiclescash ) as avgCash
FROM
    cls.mta
group by 1;


breakdown_flag       count    avgcash
-----------------  -------  ---------
Between 5 and 400   824717  153.548
Less than 5          10778    1.22323
More than 400       330233  618.516
```

This creates categories of data, based on vehiclescash and then returns how many rows are in each category.

- We can define a column by almost anything and then group by it. In the following example, we look at the difference between the vehicles which pay cash and which pay by EZ pass. If the difference is sufficiently large we categorize it one way and if not, another, but we remove zero's first!

```
select
    case
        when vehiclescash = 0 then 'Zero Cash'
        when abs( vehiclescash - vehiclesez)::float
            / vehiclescash < .05 then 'less than'
        else 'more'
    end
    , count(1) as ct
from
    cls.mta
group by 1;


case            ct
---------  -------
Zero Cash     6512
less than     2765
more       1156451
```

In the case above this returns 3 rows and 3 columns since we are aggregating on a column which can take one of three values. Aggregating on case statements is an incredibly powerful way to calculate statistics on

5 Named Subqueries

- Let's calculate the number of cars that go through each plaza each day in the Inbound direction using cash:

```
select
    sum( vehiclescash ) as totalcash
    , mtadt
    , plaza
from
    cls.mta
where
    direction = 'I'
group by mtadt, plaza;


  totalcash  mtadt         plaza
-----------  ----------  -------
      14783  2010-01-01        1
       8965  2010-01-01        2
      17309  2010-01-01        3
       3840  2010-01-01        4
       1454  2010-01-01        5
[...]
```

- Now, let's try to calculate how many cars go through the average plaza on the average day in the inbound direction using cash. In other words, we want to take the average of the above. In this case

we can try to do the following:

```
select
    avg( sum( vehiclescash) )
from
    cls.mta
where
    direction = 'I';

ERROR: aggregate function calls cannot be nested
LINE 2: avg( sum( vehiclescash) )
```

Unfortunately we can't nest aggregation functions. To answer the question above we need to use a subquery since we need to do an aggregation *on* another aggregation.

```
select
    avg( sumcash) as avgcars
from
    (select
        sum(vehiclescash) as sumcash
        , mtadt
        , plaza
    from
        cls.mta
    where direction = 'I'
    group by mtadt, plaza
    ) as innerQ;


  avgcars
---------
  7394.15
```

To understand this query, lets start by breaking it apart and focusing on the inner query first:

```
select
    sum(vehiclescash) as sumcash
    , mtadt
    , plaza
from
    cls.mta
where direction = 'I'
group by mtadt, plaza;


  sumcash  mtadt         plaza
---------  ----------  -------
    14783  2010-01-01        1
     8965  2010-01-01        2
    17309  2010-01-01        3
     3840  2010-01-01        4
     1454  2010-01-01        5
[...]
```

The result of this inner query is a table itself wth three columns and a row for each mtadt-plaza combination. The column sumcash represents the number of cars, in total, which went through that plaza-mtadt combination using cash – which is the just the number that we want to average!

Using this table we can then take an average on it, which we do in an outer query. Importantly, when we nest queries in this fashion we have to give then a name, which we do in this case with the AS clause. As a note, just like when naming a column the AS itself is optional, though recommended.

- Lets look at another example: What percentage of the day-plaza combinations in our dataset have an inbound-to-outbound ratio of less than 90% for cash transactions? In other words, what percentage of plazas, on a given day, have more outbound traffic than inbound traffic by 10%?

  Just as before we will need to compute multiple levels of aggregation. Lets work from the inside out – first computing the number of inbound and outbound cars for each plaza-mtadt combination.

DRAFT

```
select
    sum( case when direction = 'I'
        then vehiclescash else 0 end ) as InboundCash
    , sum( case when direction = 'O'
        then vehiclescash else 0 end ) as OutboundCash
from
    cls.mta
group by
    plaza, mtadt


  inboundcash    outboundcash
-------------  --------------
        14783           13383
        14680           13903
        14049           13223
        13202           13008
        12688           12530
[...]
```

Note that we are grouping by columns which we are not selecting – which is allowable under most SQL variants. Since we don't need to know which row is associated with each plaza or mtadt, only the totals, we will not select it. Once we have this data we can then do the aggregation that we are interested in:

```
select
    sum( case when InboundCash <= .90 * OutboundCash
        then 1 else 0 end)::float
            / count(1) as pct
from
    (select
        sum( case when direction = 'I'
            then vehiclescash else 0 end ) as InboundCash
        , sum( case when direction = 'O'
            then vehiclescash else 0 end ) as OutboundCash
    from
        cls.mta
    group by
        plaza, mtadt) as innerQ;


      pct
---------
0.0303516
```

- In the cases above we were required to use a subquery because we wanted to do two levels of aggregation, which is a common problem. For example, let's say that we wanted to find the average number of rows per plaza, for rows which have more than 700 EZ pass cars. In this case we first need to do two levels of aggregation – first calculating the number of rows, per plaza, which fulfill the criteria and then averaging over the plaza – as can be seen below:

```
select
    avg( numrows) as avgrows
from
    (select
        count(1) as numrows
        , plaza
    from
        cls.mta
    where
        vehiclesEZ >= 700
    group by 2) as innerQ;


  avgrows
---------
    69457
```

In the case above the inner query only has 10 rows, one for each plaza while the outer query only returns the average.

DRAFT

DRAFT

Chapter 6

Dates and Types

DRAFT

Contents

1 Date Types 103
2 Date Functions 104
3 Hard GROUP BY problems 110

DRAFT

1 Date Types

- In this section we'll be consider how Dates are dealt with in SQL. As a reminder, Figure 6.1 has the different types that most SQL standards subscribe too.

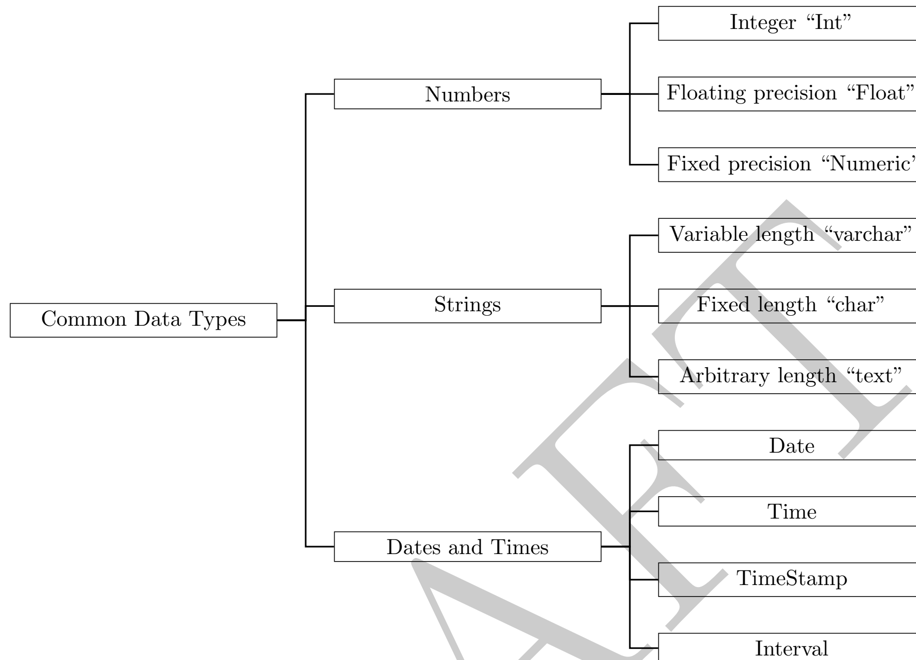


Figure 6.1: Common relational database data types

DRAFT

Dates

- Date and time functions are the *least* standardized portion of the SQL language. Different variants use different functions and conventions when in this area.
- Dates and times are *complicated*. Timezones, server and client location and configuration all yield small changes in how the database operates and what is returned in different operations.
- Date and times should not be considered "standard" between SQL variants. Different servers use different functions, types and standards. Whenever working with a variant of SQL you are unfamiliar with make sure to verify that it is doing what you expect.
- There are four standard data types in Postgres:
  1. **Dates**: Stores a date, ranging from 4713 BC to 5874897 AD.
  2. **Time**: Stores only a time with a resolution of 1 microsecond.
  3. **Timestamp**: Sometimes referred to as a "datetime." Contains both a date and a time and ranges from 4713 BC to 294276 AD.
  4. **Interval**: Time interval (such as "1 Year" or "2 hours"). These do not have a start or end and only represent a length of time.
- Timezones are problematic: Dates do not contain a timezone, but time and timestamp *may* have them. Depending on how the server and client are set-up, timezones may also prove otherwise problematic. Often it feels like timezones are applied in a haphazard way, so be careful!

- Daylight savings times, for example, starts and ends on different dates in Europe then in the United States.
- Because of all these issues, many database administrators (including myself) recommend storing times using **Unix** or **epoch** time, which is the number of seconds since 1/1/1970 in UTC. Since this is a specific point in time, there is no timezone confusion.

2 Date Functions

- There are four classes of operations we want to do with date objects:

1. **Convert a string to date:** We can do this a few different ways, but these are types of cast operators. We won't get into this too much. In simple cases we can do the "obvious" and it works.

```
select '2012/03/12'::date as dt


dt
----------
2012-03-12
```

```
select '2012/03/01 11:35:00'::timestamp as ts


ts
-------------------
2012-03-01 11:35:00
```

```
select '1 month'::interval, '2 hours'::interval as dt


interval          dt
----------------  -------
30 days, 0:00:00  2:00:00
```

   A reminder that the double colon notation is specific to Postgres. There are alternative functions, such as `to_timestamp` and `to_date` which exist in other variants which do similar things.

2. **Convert a date to a string:** This isn't something that we do that much in SQL, but if required we use the `to_char` function.

3. **Extract part of a date:** There are two functions which do this, `date_part` or `extract`. These functions extract a specific value from a date or timestamp and return it as a different type (frequently an integer). A few examples below:

```
select date_part( 'month', '2012/03/12'::date) as mnth;


 mnth
------
    3
```

```
select date_part( 'hour', '2012/03/01 11:35:00'::timestamp ) as hr;


 hr
----
 11
```

```
select extract( 'dow' from '2012/03/01 11:35:00'::timestamp) as day_of_week;


 day_of_week
-------------
           4
```

Note that `dow` starts with Sunday (at zero) and goes to Saturday (6)

4. **Basic date math and comparisons:** To do basic date math, such as adding a 3 days or subtracting an hour, we use the addition and subtraction operators with *intervals*:

```
select '2001-01-01 01:00:00'::timestamp + '21 hours'::interval as TS;


ts
-------------------
2001-01-01 22:00:00
```

Comparison operators ($<, >, =$) behave as expected. One caveat is that when you compare a timestamp with a date, the date is converted to a timestamp for midnight of that day:

```
select now() < current_date as c1, now() > current_date as c2;


c1     c2
-----  ----
False  True
```

- While the above are the four most common operations you want to do, there are some additional functions which are nice to know about:
  - `current_date` and `current_time` / `now()` return the full timestamp, the current date and the current time.

```
select now(), current_date, current_time


now                               current_date    current_time
--------------------------------  --------------  ---------------------
2023-08-14 20:50:04.440825+00:00  2023-08-14      20:50:04.440825+00:00
```

– `date_trunc`: This function truncates a date or timestamp down to a certain precision. Note that this will return a timestamp with the values beyond the specified precision set to their lowest possible value. For example:

```
select date_trunc( 'month', '2012/03/12'::date) as mnth;


mnth
-------------------------
2012-03-01 00:00:00+00:00
```

```
select date_trunc( 'hour', '2012/03/01 11:35:00'::timestamp ) as hr;


hr
-------------------
2012-03-01 11:00:00
```

In both of these examples the value being acted on loses its precision. The object returned is a timestamp with all precision below a certain threshold set to the smallest possible value.

– `date()`: Returns the date of a given timestamp in date format. This does a type conversion, which is more than simply truncating the timestamp.

```
select date(  '2012/03/01 11:35:00'::timestamp  ) as dt;


dt
----------
2012-03-01
```

Unfortunately, dates can be difficult to work with, as the following examples demonstrate:

- Seemingly arbitrary math. You can add integers to *dates*, but not to *timestamps*

```
select now() + 1 as dt;
ERROR:  operator does not exist: timestamp with time zone + integer
```

```
select date( now() ) + 1 as dt;


dt
----------
2023-08-15
```

- BETWEEN may not work as expected. For example:

  ```
  select now() between date(now()) -1 and now() as TF;


  tf
  ----
  True
  ```

  Now is between yesterday and now

  ```
  select now() between date(now()) -1 and date(now()) as TF;


  tf
  -----
  False
  ```

  but now is not between yesterday and today.

  ```
  select date(now()) between now() and date( now()) + 1 as TF;


  tf
  -----
  False
  ```

  but today is between now and tomorrow.

- Most annoying of all? Every variant of SQL is *slightly* different.
- Epoch time is pretty great at solving some of these problems, but at the cost of interpretation:
  - Math works as expected.
  - BETWEEN works as expected.
  - No time zone ambiguity.
- For the rest of this section we'll do a few time related problems using the NYC MTA data.
- Let's return the average cash volume of cars by day of the week, inbound traffic only:

```
select
    date_part('dow', mtadt ) as dow
    , avg( tvol) as avgvol
from
(select
     sum( vehiclescash ) as tvol
     , mtadt
from
    cls.mta
where
    direction = 'I'
group by 2) as innerQ
group by 1
order by 2 desc;


  dow    avgvol
-----  --------
    6   87392.9
    0   82829.7
    5   78129.6
    4   69274.6
    1   67410.7
[...]
```

- By year, what percentage of cars which pass through a toll plaza use an EZ-pass?

```
select
    date_part('year', mtadt) as yr
    , sum(vehiclesez)::float / (sum(vehiclesez) + sum( vehiclescash)) as pct_EZ
from
    cls.mta
group by 1
order by 1;


  yr    pct_ez
----  --------
2010  0.75715
2011  0.792285
2012  0.808791
2013  0.828819
2014  0.836587
[...]
```

- We can also create a time series of the number of inbound cars, by year and month, for Plaza #1 and #2:

```
select
    date_trunc('month', mtadt)
    , sum( case when plaza = 1
        then vehiclescash else 0 end) as Plaza1Cars
    , sum( case when plaza = 2
        then vehiclescash else 0 end) as Plaza2Cars
from
    cls.mta
where
    direction = 'I'
group by 1;


date_trunc                   plaza1cars    plaza2cars
-------------------------  ------------  ------------
2010-01-01 00:00:00+00:00        427660        313278
2010-02-01 00:00:00+00:00        375918        274724
2010-03-01 00:00:00+00:00        462078        354619
2010-04-01 00:00:00+00:00        455395        353378
2010-05-01 00:00:00+00:00        487051        378600
[...]
```

- When moving in and out of date formats, you may have to rely on using special functions. Part of the reason for this is because date and times require the user to specify the format. Consider the following query:

```
select
    mtadt
    , hr
    , to_timestamp( mtadt::varchar || ' ' ||  hr, 'YYYY-MM-DD HH24') as mta_ts
from cls.mta;


mtadt         hr  mta_ts
----------  ----  -------------------------
2013-10-14    16  2013-10-14 16:00:00+00:00
2013-10-14    16  2013-10-14 16:00:00+00:00
2013-10-14    17  2013-10-14 17:00:00+00:00
2013-10-14    17  2013-10-14 17:00:00+00:00
2013-10-14    18  2013-10-14 18:00:00+00:00
[...]
```

This query returns three columns: the date, hour and then it creates a timestamp object using the command `to_timestamp`. This command takes in two strings. The first is a value to be converted and the second is the format of that conversion. In this example we create a synthetic string made up of the values of mtadt concatenated with a space and then the hour. This is passed to the command and is then converted to a timestamp.

3 Hard GROUP BY problems

In this section we will look at some difficult GROUP BY problems using the MTA data as well as the stocks data sets.

1. How many stocks (symbols) have 19 or more trading days for every month in 2010?

```
select
    count(1) as ct
from
    (select
        symb
        , count(1) as ct2
    from
        (select
            symb
            , date_part( 'month', retdate) as mn
            , count(1) as ct
        from
            stocks.s2010
        group by
            1,2) as innerQ
    where
        ct >= 19
    group by 1 ) as outerQ
where ct2 = 12;


  ct
----
3106
```

2. Write a query which returns 12 rows and two columns. The first column should be month as an integer and the second should be the number of trading days in that month. Do this for 2010 and remember that dates only appear in the stocks table if they are trading days.

DF

```
select
    date_part('month', retdate) as mn
    , count( distinct retdate) as trading_days
from
    stocks.s2010
group by 1;


  mn    trading_days
----  --------------
   1             19
   2             19
   3             23
   4             21
   5             20
[...]
```

3. Create a table with the information above, this time in a wide format: one column per month with a single row.

```
select
        count( distinct case when date_part( 'month', retdate) = 1
                then retdate else null end) as Jan
        , count( distinct case when date_part( 'month', retdate) = 2
                then retdate else null end) as Feb
...[OTHER MONTHS OMITTED]
        , count( distinct case when date_part( 'month', retdate) = 12
                then retdate else null end) as Dec
from
        stocks.s2010;
```

4. Write a query which returns 12 rows and 3 columns from the 2010 data. The first column should be month as an integer, the second should be the number of unique stocks which had an open over $100 that month and the third should be the number of unique stocks with an open less than $50 that month.

DRAFT

```
select
    date_part('month', retdate) as mn
    , count( distinct case when opn > 100
        then symb else null end ) as over100
    , count( distinct case when opn < 50
        then symb else null end ) as less50
from
    stocks.s2010
group by 1;


  mn    over100    less50
----  ---------  --------
   1         68      2929
   2         58      2947
   3         65      2924
   4         71      2925
   5         68      2989
[...]
```

5. Repeat the above, but this time only include those stocks which are *also* in 2011.

```
select
    date_part('month', retdate) as mn
    , count( distinct case when opn > 100
        then symb else null end ) as over100
    , count( distinct case when opn < 50
        then symb else null end ) as less50
from
    stocks.s2010
where
    symb in (select distinct symb from stocks.s2011)
group by 1;


  mn    over100    less50
----  ---------  --------
   1         68      2904
   2         58      2924
   3         65      2901
   4         71      2903
   5         68      2969
[...]
```

6. We define the *yearly spread* as the difference between the maximum closing price for a stock and the minimum closing price for a stock over the year. Write a query which returns all stocks whose yearly spread in 2010 is less than $\frac{1}{2}$ the largest yearly spread (from all stocks) in 2011.

```
select
    symb
from
    (select max(cls) - min(cls) as ys2010
        , symb from stocks.s2010 group by 2) as IQ
where
    ys2010 < .5 * (select max(ys2011) as max_ys2011 from
        (select max(cls) - min(cls) as ys2011
            , symb from stocks.s2011 group by 2) as IQ2);


symb
------
A
AA
AAME
AAN
AAON
[...]
```

7. For stocks in 2010 return the following: (1) symbol, (2) month and (3) the difference between the maximum closing price and minimum closing price for each month. Only include those stocks which were traded more than 10 days that month.

```
select
    symb, mn, diff
from
    (select
        symb
        , date_part('month', retdate) as mn
        , max(cls) - min(cls) as diff
        , count(1) as ct
    from
        stocks.s2010
    group by 1,2 ) as innerQ
where
     ct > 10;


symb       mn    diff
------   ----   ------
A           1   2.339
A           2   1.7096
A           3   1.8097
A           4   2.382
A           5   4.0988
[...]
```

8. Return the data in the previous problem in a wide format – one column per month and one row per symbol.

```
select
        symb
        , sum( case when mn = 1 then diff else null end ) as Jan
        , sum( case when mn = 2 then diff else null end ) as Feb
[OTHER MONTHS OMITTED]
        , sum( case when mn = 12 then diff else null end ) as Dec
from
        (select
                symb
                , date_part('month', retdate) as mn
                , max(cls) - min(cls) as diff
                , count(1) as ct
        from
                stocks.2010
        group by 1,2) as innerQ
WHERE ct > 10
GROUP BY 1;
```

DRAFT

Chapter 7

Averages

DRAFT

Contents

1 The Trouble with Averages 117
2 HAVING 119
3 COALESCE and NVL 120

DRAFT

1 The Trouble with Averages

- A common difficulty of working with data is being precise when asking a question. For example, consider that you have a table which contains information from a bank. Each row contains monthly information about a bank customer, including their balance, if they own a home and some other demographic information. We could, seemingly, answer the following questions:

  – What is the average bank account size for people with more than $2,500 in their account?

  – What is the average bank account size for people who own their own home?

  However, these questions *aren't* that well-defined since bank customers move through time and their characteristics change. What happens to a customer who sells their home in the middle of the year – Do you include them in the second question above? What if a person's bank balance changes over time – do you include them in the first question?

- Let's consider a specific example:

  "What is the average number of rows per plaza with a vehicles cash per hour greater than 500?"

  In this case, there are a number of different ways that we can answer the question:

  1. **Any Row:** We only include those rows which have a vehicles cash per hour greater than 1,000:

```
select avg( ct ) as avgct
from
    (select
        count(1) as ct
    from
        cls.mta
    where vehiclescash > 500
    group by plaza ) as innerQ;


  avgct
-------
  22432
```

  2. **One Time:** We include all information from a plaza if it ever crosses the boundary:

DRAFT

```
select avg( ct ) as avgct
from
    (select
        max( vehiclescash) as maxcash
        , count(1) as ct
        , plaza
    from
        cls.mta
    group by plaza ) as innerQ
where maxcash > 500;


  avgct
-------
 116573
```

3. **Always:** We include all counties which always have a vehiclescash greater than 500:

```
select avg( ct ) as avgct
from
    (select
        min( vehiclescash) as mincash
        , count(1) as ct
        , plaza
    from
        cls.mta
    group by plaza ) as innerQ
where mincash > 500;


avgct
-------
```

   Note that this does not return anything since no plaza fulfills this critera

4. **On Average:** We can include all counties which have, on average, vehicles cash greater than 1,000:

```
select avg( ct ) as avgct
from
    (select
        avg( vehiclescash) as avgcash
        , count(1) as ct
        , plaza
    from
        cls.mta
    group by plaza ) as innerQ
where avgcash > 500;


  avgct
-------
  92232
```

The three previous queries are all equally correct interpretations of the question above. Since the question did not adequately define the terms used reasonable people can come to different answers. The moral of the story is that SQL requires a level of precision not generally found when discussing data. Be careful!

2 HAVING

- If we just want to return the total number of rows for each county that fulfills the previous three conditions? To do this we can use the HAVING clause, which works like a WHERE clause, but is evaluated *after* the GROUP BY. It uses the same column groupings as the GROUP BY clause.

  The HAVING clause is written after the GROUP BY, but before LIMIT, if there is a LIMIT.

  Let's return the raw data from some of the examples above:

  1. **One Time:** We include all information from a plaza if it ever crosses the boundary:

```
select
    max( vehiclescash) as maxcash
    , count(1) as ct
    , plaza
from
    cls.mta
group by plaza
having max(vehiclescash) > 500;


  maxcash      ct    plaza
---------  ------  -------
     1352  122976        1
     1040  122976        2
     1594  122976        3
     1368  120624        4
      674  122976        5
[...]
```

DRAFT

2. **Always:** We include all counties which always have a vehiclescash greater than 500:

```
select
    plaza
    , count(1) as ct
from
    cls.mta
group by plaza
having min(vehiclescash) > 500;


plaza    ct
-------  ----
```

Note that in this example we did not explicitly include the MIN in the SELECT statement.

3. **On Average:** We can include all counties which have, on average, vehicles cash greater than 1,000:

```
select
    plaza,
    count(1) as ct
from
    cls.mta
group by plaza
having avg(vehiclescash) > 500;


  plaza      ct
-------  ------
      3  122976
     11   61488
```

- In each of the examples above, only those counties which fulfill the aggregation criteria set forth in the HAVING clause are returned.

3 COALESCE and NVL

- There is a special CASE statement that is frequently used to handle null values, called COALESCE.[1] COALESCE returns the first non-Null value it encounters. Consider the following example data:

Table 7.1: Table with missing values: tab_missing

SID	phone1	phone2
1	(111) 123 4567	(222) 123 4567
2		(333) 123 4567
3	(444) 123 4567	
4		

We could run the following queries on this:

[1] In Oracle the statement is NVL.

```
select
    coalesce( phone1, phone2) as phone, SID
from
    tab_missing;

    phone      |  SID
---------------+------
(111) 123 4567 | 1
(333) 123 4567 | 2
(444) 123 4567 | 3
               | 4
```

You can see that for each row the query returns the first non-null value it finds. For the fourth SID, however, all values are Null and a Null is returned.

DRAFT

DRAFT

Chapter 8

Joins

DRAFT

Contents

1 Joins **125**

2 UNION and UNION ALL **132**

3 Best Practices when Combining Tables **134**

4 Intermediate Joins **136**

4.1 Aggregations on-self 136

4.2 Cross Joins for missing values 137

5 Statistical Analysis in SQL **138**

DRAFT

1 Joins

In this section we combine tables using the JOIN operator. There are a number of different ways to combine data, which we will go into now.

- Lets consider the following two tables, which we will use to demonstrate the different types of joins:

Table 8.1: Join Example Tables

Table 8.2: *Class1* Table

sname	grade
John	A
Jim	A
Kyle	C

Table 8.3: *Class2* Table

sname	grade
John	A
Jim	B
Ashley	F

- The first join we will consider is the LEFT JOIN, which keeps all records from the first table (the "Left Hand Side" or "LHS") and only those records that match from the second table (or the "Right Hand Side" or "RHS"):

```
select
    class1.*, class2.*
from
    cls.class1
left join
    cls.class2
on class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
Kyle     C
```

  There are two components to the JOIN syntax. The first, within the FROM clause, specifies the type of JOIN (in this case "LEFT JOIN") to attempt and the second, the ON clause, determines how two rows are defined to match. The ON operator acts like a WHERE clause in that any boolean condition or set of conditions can be put in it by using parenthesis, AND and OR. Any type the expression within the ON operator is true, the database regards those rows as matching. Mentally, you should think of a JOIN as going through every possible combination of rows and deciding if a row matches with another based on the match criteria in the ON clause.

  In this SELECT statement we choose all columns from both class1 and class2 tables. Since the columns have the same names in both tables we see that the column names are repeated in the resulting table.

  This LEFT JOIN leaves the second sname and grade Null for the "Kyle" row from the first table, as there is no matching row in the second table.

- There is also a RIGHT JOIN, which, similar to the LEFT JOIN, keeps all rows from the right-hand, or second, table:

```
select
    class1.*, class2.*
from
    cls.class1
right join
    cls.class2
on class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
                  Ashley   F
```

- If we only want to consider rows that are in *both* tables, we use an INNER JOIN, the syntax for which is just JOIN:

```
select
    class1.*, class2.*
from
    cls.class1
join
    cls.class2
on class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
```

  In this case, only John and Jim are returned since they are the only individuals that are in both tables.

- A FULL JOIN (sometimes called an OUTER JOIN, or FULL OUTER JOIN) includes all rows from either table:

DRAFT

```
select
    class1.*, class2.*
from
    cls.class1
full join
    cls.class2
on
    class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
Kyle     C
                  Ashley   F
```

Because “Kyle” <> “Ashley”, these two rows are kept separate.

- In a few instances we may wish to create every possible combination of rows[1], which we call a CROSS JOIN. The syntax for a CROSS JOIN is below:

```
select
    class1.*, class2.*
from
    cls.class1
cross join
    cls.class2;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
John     A        Jim      B
John     A        Ashley   F
Jim      A        John     A
Jim      A        Jim      B
[...]
```

Note that this returns every possible combination of rows. Since we selected “*” it also returns every column – which means columns with duplicated names. If we wanted to only return a few of the columns we could do the following:

[1] This is sometimes called a Cartesian product.

```
select
    class1.sname as name1
    , class2.sname as name2
    , class1.grade as grade1
    , class2.grade as grade2
from
    cls.class1
cross join
    cls.class2;


name1    name2    grade1    grade2
-------  -------  --------  --------
John     John     A         A
John     Jim      A         B
John     Ashley   A         F
Jim      John     A         A
Jim      Jim      A         B
[...]
```

- If the columns that we are matching on have the same name than we can use USING, rather than ON to specify the matching column. Doing so generates a different type of output:

```
select
    *
from
    cls.class1
full join
    cls.class2
USING( sname);


sname    grade    grade
-------  -------  -------
John     A        A
Jim      A        B
Kyle     C
Ashley            F
```

In this example, the database combined the sname column into a single column! USING tells the database that the columns represent the same data and need not be repeated. This type of "natural" join is extremely powerful when you are joining two tables which represent similar data.

- The following query demonstrates USING with multiple columns:

```
select
    *
from
    cls.class1
inner join
    cls.class2
using( sname, grade);


sname    grade
-------  -------
John     A
```

- Let's look at what the following returns, which uses both a USING with multiple columns and a FULL JOIN:

```
select
    *
from
    cls.class1
full join
    cls.class2
using( sname, grade);


sname    grade
-------  -------
John     A
Jim      A
Kyle     C
Jim      B
Ashley   F
```

  In this example, the database returns 5 rows, since the only row that matches on both sname and grade is John. People should study more.

- What does the following return?

```
select * from
    cls.class1
left join
    cls.class2
on class1.sname = class2.sname
and class1.grade >= class2.grade;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A
Kyle     C
```

Since this is a LEFT JOIN, this will include all values from the left hand table, but it will only match those which are alphabetically earlier or the same as the right hand side table. In other words, this returns only those rows from the right hand side where the person did better in class1.

```
 sname | grade | sname | grade
-------+-------+-------+-------
 Jim   | A     |       |
 John  | A     | John  | A
 Kyle  | C     |       |
(3 rows)
```

- Keep in mind that the USING clause creates a synthetic column using a similar construct as a CASE statement:

```
select
    sname as from_using
    , class1.sname as lhs
    , class2.sname as rhs
    , CASE
        when class1.sname is not null then class1.sname
        else class2.sname
      END as coal
from
    cls.class1
full join
    cls.class2
using( sname );


from_using    lhs    rhs     coal
------------  -----  ------  ------
John          John   John    John
Jim           Jim    Jim     Jim
Kyle          Kyle           Kyle
Ashley               Ashley  Ashley
```

The column "from_using" and "coal" are created as the output of the using statement and from the coalesce statement; from the above they are clearly the same. Importantly, the above statement also demonstrates that in the SELECT statement there is still access to the underlying, original columns.

- In the above examples we used ON to tell the database which columns to match. However, we can also use the WHERE clause to match. For example:

```
select
    class1.*, class2.*
from
    cls.class1
cross join
    cls.class2
where class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
```

  generates the same output as our inner join. In this case, we used a cross join to generate all possible combinations of rows and only kept those rows where the sname was the same in both columns via a WHERE clause.

- The following syntax is also used when doing cross joins.

```
select
    *
from
    cls.class1, cls.class2
where class1.sname = class2.sname;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
Jim      A        Jim      B
```

  This can also be done with more than two tables.

- We can also combine ON and WHERE:

```
select
    class1.*, class2.*
from
    cls.class1
left join
    cls.class2
on
    class1.sname = class2.sname
where
    class1.grade = class2.grade;


sname    grade    sname    grade
-------  -------  -------  -------
John     A        John     A
```

The above example uses a left join and a where clause to recreate an inner join.

- Very importantly, we can use any boolean expression as our JOIN condition as in the following example where we create a table which only contains those rows, in class1, which are not the same name! **Note that this joins class1 on itself!**

```
select
    lhs.sname as lname, lhs.grade as lgrade, rhs.*
from
    cls.class1 as lhs
left join
    cls.class1 as rhs
on lhs.sname <> rhs.sname;


lname    lgrade    sname    grade
-------  --------  -------  -------
John     A         Jim      A
John     A         Kyle     C
Jim      A         John     A
Jim      A         Kyle     C
Kyle     C         John     A
[...]
```

2 UNION and UNION ALL

- Another way to combine data is using UNION and UNION ALL. While the JOIN syntax puts tables side-by-side, the UNION and UNION ALL tables stack tables vertically on each other – appending (or concatenating) the data vertically.
- The syntax for UNION and UNION ALL looks a bit different than the syntax for other SQL commands since they behave not on *tables*, but on queries. Consider the following example of the UNION ALL command:

```
select sname from cls.class1
UNION ALL
select sname from cls.class2


sname
-------
John
Jim
Kyle
John
Jim
[...]
```

- The columns have to be selected in the correct order. The following query, which switches the order of grade and sname in the second table will not return properly aligned columns.

```
select sname, grade from cls.class1
UNION ALL
select grade, sname from cls.class2


sname    grade
-------  -------
John     A
Jim      A
Kyle     C
A        John
B        Jim
[...]
```

- The column types are also defined by the first statement in the command, so all SELECT statements must generate compatible columns. For example if our first query had an integer in the first column position then the second query can't put a string in the first position.
- The difference between UNION and UNION ALL is that UNION will automatically deduplicate records. For example, consider the following query:

```
select sname from cls.class1
UNION
select sname from cls.class2


sname
-------
Kyle
John
Ashley
Jim
```

In this case only four rows are returned since John and Jim are duplicates. UNION removes whole,

exact, row duplicates. Every column in the row must be the same for UNION to decide two rows are duplicates.

- Keep in mind that using UNION is very expensive as removing duplicates is a costly process.[2]
- We can use the UNION command to determine the best grade that a student received, as in the following query.

```
select sname, MIN(grade) as best_grade
from
    (select sname, grade from cls.class1
    UNION ALL
    select sname, grade from cls.class2 ) as innerQ
group by 1;


sname    best_grade
-------  ------------
Kyle     C
Jim      A
Ashley   F
John     A
```

- As we said above, both UNION and UNION ALL work on *statements* not tables, meaning that the following command will *not* complete successfully:

```
select sname, MIN(grade) as best_grade
from
    (select sname, grade from cls.class1) as lhs1
    UNION ALL
    (select sname, grade from cls.class2) as lhs2
group by 1;
```

3 Best Practices when Combining Tables

1. **Always have a Unique Side**. When you join two tables on a particular column, make sure that the column that you are joining on is unique on one side. If you join on a column with duplicates on both sides the database is going to create rows (via the cartesian product), a generally negative outcome.

    - Up to this point we only considered the case where both sides are unique. Let's assume that are tables now look like the below:

      In the tables above, there are multiple observations for the name "John". The lack of uniqueness causes problems, as we will see in the following query:

[2] On my computer, using the NYSE dataset, it took 2 seconds to count the number of rows after a UNION ALL between 2010 and 2011 while doing a UNION took over five times as long.

DRAFT

Table 8.4: Join Example Tables (II)

Table 8.5: *Class3* Table

sname	grade
John	A
John	B
Kyle	C

Table 8.6: *Class4* Table

sname	grade
John	A
John	B
John	C
Tim	F

```
select
    class3.sname as lname
    , class3.grade as lgrade
    , class4.sname as rname
    , class4.grade as rgrade
from
    cls.class3
left join
    cls.class4
on class3.sname = class4.sname;


lname    lgrade    rname    rgrade
-------  --------  -------  --------
John     A         John     A
John     A         John     B
John     A         John     C
John     A         John     A
John     A         John     B
[...]
```

- In this case, we can see that every pair-wise combination of the matching rows ($6 = 3 \cdot 2$ for John) was generated by the query. In other words, the "ON" clause behaved as if a "WHERE" clause; each time a row matched it was returned.
- Another way of thinking about this is that when the database encounters multiple matching rows it behaves similar to a cross-join.

2. When using JOIN, label each of the tables that you are joining on based either on:

   (a) Their location or position ("LHS", "RHS", etc.)

   (b) Their contents/the data that they contain

   Naming tables in this way leads to increased readability.

3. In terms of efficiency, joins should be undertaken in the following order:

   (i) inner

   (ii) left

   (iii) outer

   (iv) cross

DI

There are two major reasons for this order: (1) Readability (we read left to right and combining left and right joins creates difficult to understand queries) and (2) Query optimization (which we will touch upon later).

4. Be consistent with USING/ON/WHERE. I recommend using WHERE for filtering conditions, ON for matching and USING only if it makes sense. Mixing and matching yields difficult to understand queries.
5. No Nulls in join columns. Nulls do not match each other, so joining on a null always returns false! In other words, if you do a left join, the Nulls on the left are kept while the Nulls on the right are dropped.

4 Intermediate Joins

In this section we examine two common patterns around joins: (1) using cross joins to find missing data and (2) using joins with an aggregation to create datasets.

4.1 Aggregations on-self

- Consider trying to figure out what percentage of cars use EZ pass, outbound, by hour of the day. In other words we want to calculate the total number of cars which use EZ pass outbound each hour, take the sum and then divide each row. In order to do this we use a join:

```
SELECT
    hr, perhr::float / tot as pct
FROM
    (select sum(vehiclesez) as perhr, hr from cls.mta
    where direction = 'I' group by 2) as lhs
CROSS JOIN
    (select sum(vehiclesez) as tot from cls.mta
    where direction = 'I') as rhs


  hr         pct
----  ----------
   0  0.0155521
   1  0.00869534
   2  0.00568274
   3  0.00519111
   4  0.0080607
[...]
```

- What if we calculate this percentage, but by plaza? In this case we do a similar operation, but we now we join based on plaza:

```
SELECT
    plaza, hr, perhr::float / tot as pct
FROM
    (select sum(vehiclesez) as perhr, hr, plaza from cls.mta
    where direction = 'I' group by 2,3) as lhs
JOIN
    (select sum(vehiclesez) as tot, plaza from cls.mta
    where direction = 'I'
    group by plaza) as rhs
USING(plaza);


  plaza    hr        pct
-------  ----  ---------
      1     0  0.019748
      2     0  0.0135566
      3     0  0.0181986
      4     0  0.005301
      5     0  0.0143782
[...]
```

4.2 Cross Joins for missing values

- A CROSS JOIN can be useful when looking for missing data or trying to fill-in data. Let's consider the case where we want to verify that there is no missing data within the MTC table. In order to do this we can create a synthetic table which should have all values:

```
select * from
(select distinct mtadt from cls.mta ) as lhs
cross join
(select distinct hr from cls.mta ) as rhs1
cross join
(select distinct plaza from cls.mta ) as rhs2
cross join
(select distinct direction from cls.mta) as rhs3


mtadt          hr    plaza  direction
----------  ----  -------  -----------
2016-08-06    11       11  O
2016-08-06    11       11  I
2016-08-06    11        8  O
2016-08-06    11        8  I
2016-08-06    11        9  O
[...]
```

Each of the subqueries above contains the unique values for the particular column and cross joining them creates a dataset containing every possible combination of the three columns. We can then join this back against the original data to see if there are any missing values:

```
select rhs2.plaza, count(1)
    from
(select distinct mtadt from cls.mta ) as lhs
    cross join
(select distinct hr from cls.mta ) as rhs1
    cross join
(select distinct plaza from cls.mta ) as rhs2
    cross join
(select distinct direction from cls.mta) as rhs3
    left join cls.mta
using(mtadt, hr, plaza, direction)
    where mta.mtadt is null
group by 1
order by count(1) desc;


  plaza    count
-------  -------
     11    61536
      4     2400
      8      240
      5       48
      1       48
[...]
```

From this we can see that there are a number of missing observations and the plazas which are missing!

5 Statistical Analysis in SQL

In this section we will calculate a number of different features of the stock data.

1. Calculate the variance of the closing price of each stock for the year 2010. In particular, write a query which returns one row per stock and two columns: the symbol and the estimated variance of the closing price.

    - In this case we will use the formula that the variance of a variable is equal to:

    $$\begin{aligned} VAR[X] &= E[(X - \bar{X})^2] \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2 \end{aligned}$$

      which we will estimate over our data.[3]

    - The difficult part of computing this value is that we need to make sure that on each row of our dataset is the appropriate $\bar{X}$, which we deal with by calculating it in a separate query and then joining back on the original stocks data, as can be seen in the query below:

[3] While some authors divide by $n-1$ rather than n in the formula, we will stick with n as it makes only a small difference in our numbers and changing to $n-1$ can be easily accomplished using the same method.

```
select
    lhs.symb, avg( (rhs.cls - avg_cls)^2) as est_var
from
    (select avg(cls) as avg_cls, symb
        from stocks.s2010 group by 2) as lhs
join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
group by 1;


symb       est_var
------  ----------
A        6.00884
AA      22.6683
AAME     0.0532238
AAN      3.29018
AAON     0.423454
[...]
```

2. Stocks can have very large order of magnitude differences in key characteristics, such as volume and price. In order to complete analysis on these values, statisticians often normalize the values. One such normalization is the Z-score, which involves taking each value, subtracting off its mean and dividing by the standard deviation:

$$Z_i = \frac{x_i - \bar{x}}{\sigma_x}$$

where $\bar{x}$ is the mean and σ_x is the standard deviation of the variable in question.

Another method of normalization is linear, where the minimum takes on the value 0 and the maximum takes on the value of 1. This linear transformation can be computed as follows:

$$L_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- Let's calculate the linear normalization of the closing price for each stock individually. Specifically I want to return the date, symbol, closing price and normalized closing price for each stock.
- Just like in the previous example, we need to compute an aggregate (in this case, the minimum and maximum values of the closing price and join it back to the original data. Once this is complete we can then apply the formula.

DRAFT

```
select
    lhs.symb, retdate, rhs.cls,
        (rhs.cls - lhs.min_cls) / (lhs.max_cls - lhs.min_cls) as n_cls
from
    (select max(cls) as max_cls, min(cls) as min_cls, symb
        from stocks.s2010
        group by symb) as lhs
join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
order by 1,2,3;


symb    retdate         cls     n_cls
------  ----------  -------  --------
A       2010-01-04  22.3891  0.289632
A       2010-01-05  22.1459  0.26689
A       2010-01-06  22.0672  0.259531
A       2010-01-07  22.0386  0.256857
A       2010-01-08  22.0315  0.256193
[...]
```

- Note that running the above query will fail! Why? Because some stocks have min and max closing prices which are equal. In order to avoid this, we can remove those rows:

```
select
    lhs.symb, retdate, rhs.cls,
        (rhs.cls - lhs.min_cls) /(lhs.max_cls - lhs.min_cls) as n_cls
from
    (select max(cls) as max_cls, min(cls) as min_cls, symb
        from stocks.s2010
        group by symb) as lhs
join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
where min_cls != max_cls
order by 1,2;


symb    retdate         cls     n_cls
------  ----------  -------  --------
A       2010-01-04  22.3891  0.289632
A       2010-01-05  22.1459  0.26689
A       2010-01-06  22.0672  0.259531
A       2010-01-07  22.0386  0.256857
A       2010-01-08  22.0315  0.256193
[...]
```

3. Calculate the β and α coefficients of a simple linear regression of price on volume.

- As a reminder, if we run a simple linear regression of the form

$$y = \alpha + \beta x$$

, then our estimated coefficients are equal to:

$$\hat{\beta} = \frac{COV(X,Y)}{VAR(X)}$$
$$= \frac{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2}$$
$$\hat{\alpha} = \bar{Y} - \hat{\beta}\bar{X}$$

- We can calculate this using the following query:

```
select
    beta, acls - beta * avol as alpha
from
    (select
        avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) as beta
        , max( avol ) as avol
        , max (acls) as acls
    from
        (select avg(cls) as acls, avg(vol) as avol
            from stocks.s2010 ) as lhs
    cross join
        stocks.s2010 ) as IQ;


       beta    alpha
-----------  -------
0.000754955  1571.76
```

- Alternatively, we could be a bit more clever to remove the outer query:

```
select
    avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) as beta
    , max(acls) - avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) * max(avol) as alpha
    , max( avol ) as avol
    , max (acls) as acls
from
    (select avg(cls) as acls, avg(vol) as avol
        from stocks.s2010 ) as lhs
    cross join
        stocks.s2010;


       beta    alpha         avol     acls
-----------  -------  -----------  -------
0.000754955  1571.76  1.49898e+06  2703.42
```

This calculation retains the same information as the previous, but avoids using an inner query.

4. We can also calculate the R^2 for this regression using the following formula:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$
$$= 1 - \frac{\sum_{i=1}^{n}(y_i - (\alpha + \beta x_i))^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

- To complete this, we start with the previous query which generated alpha and beta and modify it to keep both the average volume and average closing price. We then CROSS JOIN this single row against the original stocks data:

```
select
    max( alpha) as alpha
    , max( beta) as beta
    , 1 - avg( (cls - (alpha + beta * vol))^2) /avg( (cls - acls)^2) as r2
from
(select
    avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) as beta
    , max(acls) - avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) * max(avol) as alpha
    , max( avol ) as avol
    , max (acls) as acls
from
    (select avg(cls) as acls, avg(vol) as avol
        from stocks.s2010 ) as lhs
    cross join
        stocks.s2010) as lhs
cross join
    stocks.s2010 as rhs;


  alpha         beta          r2
-------  -----------  ----------
1571.76  0.000754955  0.00121957
```

- Sadly our results are quite poor. The R^2 that I get is equal to 0.00122.

5. Why don't we repeat the analysis, this time *by stock*?

    - Let's first compute our α and $\hat{\beta}$ by stock:

```
select
    lhs.symb
    , avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) as beta
    , max(acls) - avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) * max(avol) as alpha
    , max( avol ) as avol
    , max (acls) as acls
from
    (select symb, avg(cls) as acls, avg(vol) as avol, count(1) as ct
        from stocks.s2010 group by 1) as lhs
    left join
        stocks.s2010 as rhs
    on lhs.symb = rhs.symb
    where ct > 100
   group by 1;


symb           beta     alpha             avol      acls
------  ------------  --------  ----------------  --------
ABC     -3.18636e-08  30.316         3.63933e+06  30.2001
CLF     -1.22468e-06  67.6353        5.69978e+06  60.6549
SRDX    -4.75616e-06  16.2657   107476            15.7546
PDLI    -3.88147e-10   6.00289       2.73978e+06   6.00183
VIAB    -3.80486e-07  35.7868        4.41476e+06  34.107
[...]
```

    notice that the query above added an additional filter to remove stocks with less than 100 data points. There are about 250 total trading days in our dataset, so by adding this filter we remove those stocks which are only in the data a small number of times. This also avoid any potential divide by zero error.

    - We can also compute our r^2 by stock, as demonstrated below. Once again, we limit ourselves to stocks which have more than 100 data points.

```
select
    lhs.symb
    , max( alpha) as alpha
    , max( beta) as beta
    , 1 - avg( (cls - (alpha + beta * vol))^2) / avg( (cls - acls)^2) as r2
from
    (select
        lhs.symb
        , avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) as beta
        , max(acls) - avg( (cls - acls) * (vol - avol) ) / avg( (vol - avol)^2) * max(avol) as alpha
        , max( avol ) as avol
        , max (acls) as acls
    from
        (select symb, avg(cls) as acls, avg(vol) as avol, count(1) as ct
            from stocks.s2010 group by 1) as lhs
        left join
            stocks.s2010 as rhs
        on lhs.symb = rhs.symb
        where ct > 100
    group by 1) as lhs
left join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
group by lhs.symb
order by r2 desc;


symb       alpha          beta        r2
------  --------  ------------  --------
HOV       3.4266    3.46633e-07  0.612527
MGIC     2.37857    1.78381e-06  0.582692
HPJ      3.54329    7.42622e-06  0.55797
FTK      1.36355    8.33776e-07  0.524931
NUV     10.1132   -1.03103e-06  0.516196
[...]
```

6. Why not with scaled parameters?

```
with sd as (
select
    lhs.symb, retdate
        , (rhs.cls - lhs.min_cls) / (lhs.max_cls - lhs.min_cls) as ncls
        , (rhs.vol::float - lhs.min_vol) / (lhs.max_vol - lhs.min_vol) as nvol
from
    (select max(cls) as max_cls, min(cls) as min_cls
        , symb, max(vol) as max_vol, min(vol) as min_vol
        from stocks.s2010
        group by symb) as lhs
join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
where min_cls <> max_cls and min_vol <> max_vol
)

select
    lhs.symb
    , max( alpha) as alpha
    , max( beta) as beta
    , 1 - avg( (ncls - (alpha + beta * nvol))^2) / avg( (ncls - acls)^2) as r2
from
    (select
        lhs.symb
        , avg( (ncls - acls) * (nvol - avol) ) / avg( (nvol - avol)^2) as beta
        , max(acls) - avg( (ncls - acls) * (nvol - avol) ) / avg( (nvol - avol)^2) * max(avol) as alpha
        , max( avol ) as avol
        , max (acls) as acls
    from
        (select symb, avg(ncls) as acls, avg(nvol) as avol, count(1) as ct
            from sd group by 1) as lhs
        left join
            sd as rhs
        on lhs.symb = rhs.symb
        where ct > 100
    group by 1) as lhs
left join
    sd as rhs
on lhs.symb = rhs.symb
group by lhs.symb
order by r2 desc;


symb        alpha       beta        r2
------  ---------  ---------  --------
HOV     0.0333546   0.823546  0.612527
MGIC    0.125339    1.27091   0.582692
HPJ     0.0803212   0.976241  0.55797
FTK     0.076173    1.08304   0.524931
NUV     0.858118   -1.1852    0.516196
[...]
```

7. What about Z-scaled? STOPPED HERE.

```
with sd as (
select
    lhs.symb, retdate
        , (rhs.cls - lhs.min_cls) / (lhs.max_cls - lhs.min_cls) as ncls
        , (rhs.vol::float - lhs.min_vol) / (lhs.max_vol - lhs.min_vol) as nvol
from
    (select max(cls) as max_cls, min(cls) as min_cls
        , symb, max(vol) as max_vol, min(vol) as min_vol
        from stocks.s2010
        group by symb) as lhs
join
    stocks.s2010 as rhs
on lhs.symb = rhs.symb
where min_cls <> max_cls and min_vol <> max_vol
)

select
    lhs.symb
    , max( alpha) as alpha
    , max( beta) as beta
    , 1 - avg( (ncls - (alpha + beta * nvol))^2) / avg( (ncls - acls)^2) as r2
from
    (select
        lhs.symb
        , avg( (ncls - acls) * (nvol - avol) ) / avg( (nvol - avol)^2) as beta
        , max(acls) - avg( (ncls - acls) * (nvol - avol) ) / avg( (nvol - avol)^2) * max(avol) as alpha
        , max( avol ) as avol
        , max (acls) as acls
    from
        (select symb, avg(ncls) as acls, avg(nvol) as avol, count(1) as ct
            from sd group by 1) as lhs
        left join
            sd as rhs
        on lhs.symb = rhs.symb
        where ct > 100
    group by 1) as lhs
left join
    sd as rhs
on lhs.symb = rhs.symb
group by lhs.symb
order by r2 desc;


symb        alpha       beta        r2
------  ---------  ---------  --------
HOV     0.0333546   0.823546  0.612527
MGIC    0.125339    1.27091   0.582692
HPJ     0.0803212   0.976241  0.55797
FTK     0.076173    1.08304   0.524931
NUV     0.858118   -1.1852    0.516196
[...]
```

DRAFT

Chapter 9

Advanced Joins

DRAFT

Contents

1 The Shape of Data **149**

2 Revenue over time & Advanced Joins **151**

2.1 First Value 152

2.2 Most common value by group 156

2.3 Cumulative Sum 158

2.4 Rolling 90 day Calculation 160

2.5 Cohorted Monthly Revenue 161

DRAFT

1 The Shape of Data

- Up to this point we have taken the data given to us as a given: The columns and rows are what they are. However, it is often useful to reshape the data by interchanging rows and columns for other purposes. For example, consider the following two tables:

Table 9.1: Example of wide data: *house_wide*

owner_name	NoBedroomHouse1	NoBedRoomHouse2	CostHouse1	CostHouse2
Rick	3	2	250000	125000
Harry	2	3	250000	125000
James	1		125000	
Lenka	3		450000	

Table 9.2: Example of long data: *house_long*

owner_name	HouseNo	BedRoom	Cost
Rick	1	3	250000
Rick	2	2	125000
Harry	1	2	250000
Harry	2	3	125000
James	1	1	125000
Lenka	1	3	450000

- We would characterize the first table as being "wide" and the second as being "long." While both tables contain the same information depending on the application one shape can be easier to use than the other. Consider the following two questions:

  1. What is the average cost of a person's second house?

```
select avg( CostHouse2 ) as avg_cost from cls.house_wide;


  avg_cost
----------
    125000
```

```
select avg(Cost) from cls.house_long where HouseNo = 2;


   avg
------
125000
```

  2. What is the average cost of any house?

DRAFT

```
select
     (sum( CostHouse1 ) + sum( CostHouse2 ) )
         / (count( CostHouse1) + count(CostHouse2)) as avg_cost
from cls.house_wide;


  avg_cost
----------
    220833
```

```
select avg(Cost) as avg_cost  from cls.house_long;


  avg_cost
----------
    217500
```

- Looking at the examples above you can see that, even if the case of these simple statistics different data shapes can make a big difference. This is especially important when exporting data to another program.
- We can use GROUP BY and CASE statements to reshape data from long-to-wide:

```
select
    owner_name
    , max( case when HouseNo = 1
        then BedRoom else null end ) as NoBedroomHouse1
    , max( case when HouseNo = 2
        then BedRoom else null end ) as NoBedroomHouse2
    , max( case when HouseNo = 1
        then Cost else null end ) as CostHouse1
    , max( case when HouseNo = 2
        then Cost else null end ) as CostHouse2
from
    cls.house_long
group by 1;


owner_name      nobedroomhouse1    nobedroomhouse2    costhouse1    costhouse2
------------  -----------------  -----------------  ------------  ------------
Rick                          3                  2        230000        125000
Lenka                         3                           450000
James                         1                           125000
Harry                         2                  3        250000        125000
```

- We can use JOIN and UNION ALL to move between wide-to-long:

```
select
    lhs.owner_name
    , lhs.houseNo
    , case
        when houseNo = 1 then nobedroomhouse1
        when houseNo = 2 then nobedroomhouse2
        else null end as nbr
    , case when houseNo = 1 then costhouse1
        when houseNo = 2 then costhouse2
        else null end as ch
from
    (select distinct owner_name, 1 as houseNo from cls.house_wide
        union all
    select distinct owner_name, 2 as houseNo from cls.house_wide) as lhs
LEFT JOIN
    cls.house_wide
using(owner_name)
where case
        when houseNo = 1 then nobedroomhouse1
        when houseNo = 2 then nobedroomhouse2
    else null end  is not null;


owner_name      houseno    nbr      ch
------------  ---------  -----  ------
James                 1      1  125000
Rick                  1      3  250000
Lenka                 1      3  450000
Harry                 1      2  250000
Rick                  2      2  125000
[...]
```

- These constructs – *wide* vs. *long* are important to be able to swap between. Other programming languages often have commands like "pivot", "reshape", "rollup" or "crosstab" that generate data in different forms, sometimes with aggregations occurring.

D

2 Revenue over time & Advanced Joins

- In this section we consider a common application for reshaping data and that is calculating business statistics from transaction data.
- Consider the following dataset which contains information on a business. This contains transaction information where each row represents a particular event. In this case, the event under consideration is the purchase of special soap bars. There are two types of transactions: single bars and double bars while there are two types: "Unit" which represents a one-off transaction and "Sub" which represents a subscription.
- A very common task when analyzing transaction data is understanding the revenue generated by a customer over time. This number (sometimes called LTV or ARPU) is based on "cohorts" of users, or defined groups of users with similar characteristics.
- Using the above data, how would we calculate the average amount spent by each customer?

Figure 9.1: *Trans* table, 1,063,491 rows

orderid	userid	trans	type	locale	trans_dt	units	coupon	months	amt
0	1	Double bar	Unit	U.S.	2016-05-09	2			39.98
1	2	Single bar	Unit	U.S.	2018-07-09	3			35.97
2	2	Single bar	Unit	U.S.	2018-08-25	1			11.99
3	2	Single bar	Unit	U.S.	2018-02-16	1			11.99
4	3	Single bar	Unit	U.S.	2016-02-28	4			47.96
5	4	Double bar	Sub	Canada	2018-03-09	5	25	2	74.96
6	4	Double bar	Sub	Canada	2018-05-09	5	25	2	74.96
7	5	Single bar	Sub	Canada	2016-01-05	4	35	2	31.17
8	6	Double bar	Unit	U.S.	2017-04-13	2			39.98
9	6	Double bar	Unit	U.S.	2016-07-28	4			79.96

```
select
    sum( amt ) / count(distinct userid) as amtPerUser
from cls.trans;


  amtperuser
------------
     69.4199
```

2.1 First Value

- Let's say that we were interested in understanding how relative countries monetized, how would we calculate the amount per user for each country? In other words, if we defined the cohort based on where a user lives, how would the countries compare?

```
select
    locale
    , sum( amt ) / count(distinct userid) as amtPerUser
from
    cls.trans
group by 1;


locale      amtperuser
--------  ------------
Canada         63.2709
Mexico         72.5456
U.S.           62.5774
```

- What happens if a user moves? How is the average amount per country affected if users can move? How should we handle calculating the average amount per user per country? We would probably want to take the first one that a user appears in:

```
select
    new_locale
    , sum( amt ) / count(distinct lhs.userid) as amtPerUser
from
    (select
        min( trans_dt ) as mindt, userid
    from
        cls.trans
    group by 2) as lhs
join
    (select
        userid, locale as new_locale, trans_dt
    from
        cls.trans) as rhs
on
    lhs.mindt = rhs.trans_dt
    and lhs.userid = rhs.userid
left join
    cls.trans
on lhs.userid = trans.userid
group by 1;


new_locale      amtperuser
------------  ------------
Canada             69.3196
Mexico            106.897
U.S.               65.8775
```

Take a look at how the query works. This is an example of identifying a "first value" of a customer. In this case we first identify the column that we are interested in ordering by, identifying the row of interest and then re-joining to the original data based on that row.

- What is the total amount spent by customers by first purchase type (subscription vs. unit sale)? In order to do this we must identify what the first purchase was for each user:

Draft

```
select
    lhs.userid, trans.type
from
    (select
        userid, min(trans_dt) as firstdt
    from
        cls.trans
    group by 1) as lhs
left join
    cls.trans
on
    lhs.userid = trans.userid
    and lhs.firstdt = trans.trans_dt


  userid  type
--------  ------
       4  Units
       6  Units
       7  Sub
      10  Sub
      21  Units
[...]
```

What if we a user can make multiple purchases in a day – What do we do in this case? Lets assume that we want to prioritize Subscriptions over Units, so that if a user makes multiple purchases in a day that they are flagged as subscribers:

DRAFT

```
 select
    lhs.userid
    , max( case when trans.type = 'Sub' then 1
        else 0 end ) as subscriber_flag
    , min( firstdt) as firstdt
from
    (select
        userid, min(trans_dt) as firstdt
    from
        cls.trans
    group by 1) as lhs
left join
    cls.trans
on
    lhs.userid = trans.userid
    and lhs.firstdt = trans.trans_dt
group by 1;


  userid    subscriber_flag  firstdt
--------  -----------------  ----------
   84925                  0  2018-05-10
  165533                  0  2018-10-12
  162195                  0  2018-11-14
   47051                  0  2016-03-06
  161180                  1  2016-01-18
[...]
```

- Now that we have identified the type of user, we then need to re-remerge that back onto the data to get the rest of the information that we need:

DRAFT

```
select
    subscriber_flag
    , count(distinct outerLHS.userid) as numusers
    , sum( amt) as totalamt
    , sum(amt) / count(distinct outerLHS.userid) as avg
from
    (select
        lhs.userid
        , max( case when type = 'Sub' then 1
            else 0 end ) as subscriber_flag
        , min( firstdt) as firstdt
    from
        (select
            userid, min(trans_dt) as firstdt
        from
            cls.trans
        group by 1) as lhs
    left join
        cls.trans
    on
        lhs.userid = trans.userid
        and lhs.firstdt = trans.trans_dt
    group by 1) as outerLHS
left join cls.trans
using(userid)
group by 1;


  subscriber_flag    numusers     totalamt      avg
-----------------  ----------  -----------  -------
                0      379309  2.40611e+07  63.434
                1      194980  1.5806e+07   81.0648
```

2.2 Most common value by group

- Another very common task is to find the most common value for a particular group. For example, lets say that we want to figure out what the most common value is among a particular sub group.
- For example, what is the most common order amount (dollars) for each country?

```
select
    locale, amt, count(1)
from
    cls.trans
group by 1,2
order by 3 desc;


locale      amt    count
--------  -----  -------
U.S.      39.98    65523
U.S.      23.98    64606
U.S.      59.97    51018
U.S.      35.97    50809
U.S.      19.99    34005
[...]
```

- Looking at the query above we can see that the most common amount for the US is 39.98, while in Canada and Mexico the amounts are 25.99 and 17.98 respectively. We now want to write a query which identifies just those three values. To do this we need to take this table and join it on itself. Lets look at the following query:

```
select lhs.locale, lhs.amt, lhs.ct
from
    (select locale, amt, count(1) as ct from cls.trans
     group by 1,2) as lhs
left join
    (select locale, amt, count(1) as ct from cls.trans
     group by 1,2) as rhs
on lhs.locale = rhs.locale and lhs.ct <= rhs.ct
group by 1,2,3
having count(rhs.*) = 1;


locale      amt     ct
--------  -----  -----
Canada    25.99  24411
Mexico    35.97  21847
U.S.      39.98  65523
```

- This query works by exploding the dataset via the left join and then collapsing it down along all the left hand side variables. The join itself only matches those counts from the left hand side which are less than or equal to those on the right hand. In other words, this creates a row numbering based on the original count! If you want to see this, run the previous query while removing the final GROUP BY and HAVING.

- This technique can be used to also find the least common value (swapping the inequality to a greater than) or even the second or third highest value (how would this be done?)

2.3 Cumulative Sum

- Another common, difficult query to write is to write a cumulative sum, which adds up all values previous to and including the current row. We need to use the same technique as in the previous examples, but this time use the trans_dt field to help us order the columns:

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, amt, trans_dt from cls.trans) as lhs
left join
    (select userid, amt, trans_dt from cls.trans) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
group by 1,2,3
order by 1,3;


  userid    amt  trans_dt      cumsum
--------  -----  ----------  --------
       1  23.98  2016-05-09     23.98
       2  12.99  2018-08-25     12.99
       3  43.16  2017-03-05     43.16
       3  43.16  2017-04-05     86.32
       4  59.95  2016-02-28     59.95
[...]
```

- What if there were multiple values on a particular day?
- In the case of multiple days you the above query will actually generate data since the merge is not unique on each side! This is bad – the sum of the amount of money should be conserved, but if we generate rows the number will actually increase. So how would we get around this? We can sum up by date to make sure that each row is unique by date:

DRAFT

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
group by 1,2,3
order by 1,3;


  userid    amt  trans_dt       cumsum
--------  -----  ----------  --------
       1  23.98  2016-05-09     23.98
       2  12.99  2018-08-25     12.99
       3  43.16  2017-03-05     43.16
       3  43.16  2017-04-05     86.32
       4  59.95  2016-02-28     59.95
[...]
```

By doing this aggregation we now avoid any creating any data.

- What is we wanted to do the above, but *not* include the current date? To do this we modify the join condition:

DRAFT

```
select
    lhs.userid, lhs.amt, lhs.trans_dt
    , sum(rhs.amt) as cumsum
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt > rhs.trans_dt
group by 1,2,3
order by 1,3;


  userid    amt  trans_dt      cumsum
--------  -----  ----------  --------
       1  23.98  2016-05-09
       2  12.99  2018-08-25
       3  43.16  2017-03-05
       3  43.16  2017-04-05     43.16
       4  59.95  2016-02-28
[...]
```

2.4 Rolling 90 day Calculation

- What happens when we move into a new locale? If we calculate the average revenue using the ways described above then any new country will look terrible because it is simply younger than the other countries.
- To get rid of this issue we *always* cohort users by when they begin a service. This allows us to compare apples-to-apples, rather than biasing our analysis toward those cohorts which have had more time to matriculate within the system.
- Lets say that we wish to do a rolling calculation – say I want to calculate the average transaction size for the first three months for each customer?

DRAFT

```
select lhs.userid, lhs.trans_dt, lhs.amt, sum( rhs.amt)
from
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as lhs
left join
    (select userid, sum( amt ) as amt, trans_dt
        from cls.trans
    group by 1,3) as rhs
on lhs.userid = rhs.userid and lhs.trans_dt >= rhs.trans_dt
    and lhs.trans_dt <= rhs.trans_dt + 90
group by lhs.userid, lhs.trans_dt, lhs.amt;


  userid  trans_dt      amt    sum
--------  ----------  -----  -----
       1  2016-05-09  23.98  23.98
       2  2018-08-25  12.99  12.99
       3  2017-03-05  43.16  43.16
       3  2017-04-05  43.16  86.32
       4  2016-02-28  59.95  59.95
[...]
```

2.5 Cohorted Monthly Revenue

- For plotting purposes we often want to break down the revenue over time, by the cohort or install date.
- In the following example, we calculate this by month of first transaction and then we return the results in a wide format. Why would we return this data in a wide format? Because this allows us to plot it fairly easily.

DRAFT

```
select
    cohort::date
    , count(distinct userid) as numusers
    , sum(case when trans_dt::date
        between cohort and (cohort + '1 month'::interval)::date
        then amt else 0 end ) as mon_0_amt
    , sum(case when trans_dt::date
        between (cohort + '1 month'::interval)::date
            and (cohort + '2 month'::interval)::date
        then amt else 0 end ) as mon_1_amt
    , sum(case when trans_dt::date
        between (cohort + '2 month'::interval)::date
            and (cohort + '3 month'::interval)::date
        then amt else 0 end ) as mon_2_amt
from
    cls.trans as lhs
left join
    (select userid, date_trunc( 'month', min( trans_dt)) as cohort
    from cls.trans group by 1) as rhs
using(userid)
GROUP BY 1;


cohort        numusers    mon_0_amt    mon_1_amt    mon_2_amt
----------  ----------  -----------  -----------  -----------
2016-01-01       21302       891314      68182         132567
2016-02-01       19503       819087      65729.4       125490
2016-03-01       20339       850657      67757.5       130519
2016-04-01       19571       819085      65591.1       124968
2016-05-01       19408       812544      65243.4       125081
[...]
```

- If we wanted to do this long, we could do the following. Note that by making the data long, we don't need to have an artificial monthly cut-off:

```
select
    rhs.cohort::date
    , rhs2.newusers
    , 12* ( DATE_PART('year', trans_dt::date) - DATE_PART('year', rhs.cohort)  )
    + (DATE_PART('month', trans_dt::date) - DATE_PART('month', rhs.cohort)) as numMonths
    , sum( amt) as revenue
from
    cls.trans as lhs
left join
    (select userid, date_trunc( 'month', min( trans_dt)) as cohort
    from cls.trans group by 1) as rhs
using(userid)
left join
    (select count(distinct userid) as newusers, cohort
    from
        (select userid, date_trunc( 'month', min( trans_dt)) as cohort
        from cls.trans group by 1) as innerrhs
    group by 2) as rhs2
on rhs.cohort = rhs2.cohort
GROUP BY 1,2,3


cohort        newusers    nummonths    revenue
----------  ----------  -----------  ---------
2016-01-01       21302            0   889046
2016-01-01       21302            1    63285.4
2016-01-01       21302            2   131443
2016-01-01       21302            3    27301.3
2016-01-01       21302            4    69052.8
[...]
```

Annoyingly, look at what we had to do to get the number of new users within each cohort into the resulting data!.

DRAFT

DRAFT

Chapter 10

Analytic Functions & CTE's

DRAFT

Contents

1 Analytic Functions **167**
2 Using Analytic Functions with Transaction Data **174**
3 Common Table Expressions ("CTE") **176**
4 CTEs with the transaction data **178**

DRAFT

1 Analytic Functions

- Analytic (sometimes called "window" or "partition" functions) functions were designed to simplify many common complex joins.
- There are a few different use cases that they can greatly simplify.
- As an example, lets consider the case of computing the *percentage* of traffic which pays by cash by hour and plaza in the inbound direction on November 10th, 2016. To do this we would need to take our original data and then join it against the correct sum. In other words we want to return 24 columns for each plaza and the sum (vertically, across hour) should be equal to 1.
- In order to do this calculation we need to join our original data back onto the proper sum, as can be seen in the query below.

```
select
    plaza
    , hr
    , vehiclescash::float / ALLVech as pctperhr
    , vehiclescash
    , ALLVech
from
    (select plaza, hr, vehiclescash
        from cls.mta
        where mtadt = '2016-11-10'
        and direction = 'I') as lhs
left join
    (select plaza, sum(vehiclescash) as ALLVech
        from cls.mta
        where mtadt = '2016-11-10'
        and direction = 'I'
        group by 1) as rhs
using(plaza)
order by 1,2;


  plaza    hr    pctperhr    vehiclescash    allvech
-------  ----  ----------  --------------  ---------
      1     0  0.0230918              167       7232
      1     1  0.017146               124       7232
      1     2  0.00954093              69       7232
      1     3  0.00940265              68       7232
      1     4  0.0199115              144       7232
[...]
```

but this construct is a bit cumbersome.

- For another example, consider wanting to create a rolling sum over each plaza day for the number of cars which use cash in the inbound direction.

```
select
    lhs.plaza, lhs.mtadt, lhs.hr, sum( rhs.vehiclescash ) as cum_sum
from
    (select plaza, hr, mtadt
            from cls.mta where direction = 'I') as lhs
left join
    (select plaza, hr, mtadt, vehiclescash
            from cls.mta where direction = 'I') as rhs
on
    lhs.plaza = rhs.plaza
    and lhs.hr >= rhs.hr
    and lhs.mtadt = rhs.mtadt
group by lhs.plaza, lhs.mtadt, lhs.hr
order by 1,2,3


  plaza  mtadt         hr    cum_sum
-------  ----------  ----  ---------
      1  2010-01-01     0        474
      1  2010-01-01     1       1191
      1  2010-01-01     2       1855
      1  2010-01-01     3       2450
      1  2010-01-01     4       2997
[...]
```

- These two examples have a set of common properties: we need to aggregate over our table while returning the original table. Doing this using the techniques we've seen in the past is cumbersome, so we can use Analytic (sometimes called window or partition functions) to solve them.
- Analytic functions use the following syntax:

```
function () over(
        partition by ______________
        order by ______________
        <WINDOW FRAME CLAUSE>
)
```

function can be one of any of our standard aggregate functions (SUM, COUNT, MAX, MIN, AVG) as well as a number of functions that can only be used as analytic functions.

There are a few pieces of the syntax:

1. The OVER() clause: This tells the database to expect a window function, rather than a standard aggregate function. This is required when using analytic functions.
2. The PARTITION BY clause: This clause tells the database how to break up the data. In other words, it is similar to a GROUP BY in that it tells the database that rows with the same values should be treated as a single entity or partition. The PARTITION BY clause is optional.
3. The ORDER BY clause: This clause works just as an ORDER BY in a normal SQL query works. It tells the database how to sort the data within each partition. The ORDER BY clause is optional. If an ORDER BY clause is present then the function is calculated in a running

fashion – e.g. as a running some from the start of the partition to the current row.

4. The WINDOW FRAME clause defines the region over which the function is calculated. It takes on a number of different forms though the most common is the rows between syntax:

```
ROWS BETWEEN ______________ AND ____________
```

the blanks would take on some of the following values:

– UNBOUNDED PRECEDING: from the start of the partition

– UNBOUNDED FOLLOWING: to the end of the partition

– XX PRECEDING: XX rows preceding (inclusive)

– XX FOLLOWING: XX rows following (inclusive)

– CURRENT ROW: the current row

In other words we can use this syntax to easily compute things like hourly moving averages or just smoothing.

- Lets use analytics functions to solve the two problems at the start of this section. To solve the first one we can do the following:

```
select
    plaza
    , hr
    , vehiclescash::float/sum(vehiclescash) over(partition by plaza) as pctperhr
    , vehiclescash
    , sum( vehiclescash) over(partition by plaza) as ALLVech
from
    cls.mta
where
    mtadt = '2016-11-10' and direction = 'I'
order by 1,2;


  plaza    hr    pctperhr    vehiclescash    allvech
-------  ----  ----------  --------------  ---------
      1     0  0.0230918              167       7232
      1     1  0.017146               124       7232
      1     2  0.00954093              69       7232
      1     3  0.00940265              68       7232
      1     4  0.0199115              144       7232
[...]
```

the OVER clause, which modifies the SUM function, tells the database that it is going to be computing an aggregate function, but *without the aggregation*. In other words, it will return the same value for each row.

- To be clear on what this is doing, let's consider only a single plaza (#1) and look at the hourly data for that day, including the analytic function:

```
select
    plaza
    , hr
    , vehiclescash
    , sum(vehiclescash) over(partition by plaza) as totalcars
from
    cls.mta
where
    mtadt = '2016-11-10'
    and direction = 'I'
    and plaza = 1;


  plaza    hr    vehiclescash    totalcars
-------   ----  --------------  -----------
      1     0             167         7232
      1     1             124         7232
      1     2              69         7232
      1     3              68         7232
      1     4             144         7232
      1     5             215         7232
      1     6             281         7232
      1     7             336         7232
      1     8             329         7232
      1     9             304         7232
      1    10             344         7232
      1    11             286         7232
      1    12             308         7232
      1    13             375         7232
      1    14             361         7232
      1    15             471         7232
      1    16             450         7232
      1    17             451         7232
      1    18             420         7232
      1    19             446         7232
      1    20             366         7232
      1    21             322         7232
      1    22             296         7232
      1    23             299         7232
```

The sum of vehiclescash on this subset is 7,232 – the exact number returned by the analytic function in the total cars column.

- To solve the cumulative sum problem we can use an analytic function in the following manner:

```
select
    plaza, mtadt, hr,
    sum(vehiclescash) over(
        partition by plaza, mtadt
        order by hr
        rows between unbounded preceding and current row) as cum_sum
from
    cls.mta
where
    direction = 'I'


  plaza  mtadt          hr    cum_sum
-------  ----------  ----  ---------
      1  2010-01-01     0        474
      1  2010-01-01     1       1191
      1  2010-01-01     2       1855
      1  2010-01-01     3       2450
      1  2010-01-01     4       2997
[...]
```

- To get more insight into the specifics of how analytic functions work, consider the following table:

GRP	ORD	NM	C0	C1	C2	C3
1	1	5	65	33	5	11
1	2	6	65	33	11	21
1	3	10	65	33	21	28
1	4	12	65	33	33	22
2	1	12	65	32	12	22
2	2	10	65	32	22	28
2	3	6	65	32	28	20
2	4	4	65	32	32	10

In this table the raw data is GRP, ORD and NM. In order to create columns C1, C2 and C3 we use the following syntax:

```
SUM(NM) OVER( ) as C0
SUM(NM) OVER( PARTITION BY GRP ) as C1
SUM(NM) OVER( PARTITION BY GRP ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as C1
SUM(NM) OVER( PARTITION BY GRP ORDER BY ORD ASC) as C2
SUM(NM) OVER( PARTITION BY GRP ORDER BY ORD ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS C3
```

- **NOTE:** The default behavior of different analytic functions when using different sets of arguments can lead to issues. I have a few cases memorized, but I'd recommend being as inclusive as possible with the arguments. In the example above, `C1` is presented twice, both would return the same numbers, but the second makes it more clear as to what is happening.

- All of the aggregate functions we used in the past (MAX, MIN, COUNT and AVG) can be used with analytic functions. For example, to create a moving average based on the last 4 hours of data in the MTA dataset we could do the the following. Note that the ROWS BETWEEN function is inclusive, so it will average over four hours in the below.

```
select
    plaza
    , direction
    , hr
    , mtadt
    , vehiclescash + vehiclesez as totalcars
    , avg(vehiclescash + vehiclesez)
        OVER(PARTITION by plaza, direction
            ORDER BY mtadt asc, hr asc
            ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) as hravg
from
    cls.mta;


  plaza  direction      hr  mtadt         totalcars    hravg
-------  -----------  ----  ----------  -----------  -------
      1  I               0  2010-01-01          889    889
      1  I               1  2010-01-01         1419   1154
      1  I               2  2010-01-01         1223   1177
      1  I               3  2010-01-01         1075   1151.5
      1  I               4  2010-01-01          945   1165.5
[...]
```

DRAFT

- If we use a WINDOW FRAME clause without an ORDER BY then the row order returned is arbitrary.
- The only aggregate function not allowed is COUNT(DISTINCT) which cannot be used with the OVER() clause.
- There are a number of non-aggregate functions that can be used as analytic functions:
    1. LAG() and LEAD(): These functions return the value of a column from a preceding or following row.
    2. FIRST_VALUE(), LAST_VALUE() and NTH_VALUE() : These function return the first, last, or more generally, the nth value within a partition. Note that the NTH_VALUE function takes not only a column name, but a positional argument starting at 1. It will return null if there aren't enough values within the partition.
    3. NTILE(): This function handles percentiles.
    4. ROW_NUMBER(): This function returns the row number based on the criteria established in the clause. Note that the function ROW_NUMBER() fails without an OVER clause.
    5. RANK(): Returns the rank of a particular observation
    6. DENSE_RANK(): Returns the dense rank of a particular observation.
- The commands `ROW_NUMBER`, `RANK` and `DENSE_RANK` behave similarity when the data being sorted is unique. If the data is not unique, however, these commands behave differently, as demonstrated in the Table 10.1
- These functions allow us to easily answer a number of questions (without using JOIN). Such as what is the correlation between the absolute (nominal) change in vehicles paying cash and the absolute (nominal) change in vehicles paying by EZ-pass?

ID	ROW_NUMBER	RANK	DENSE_RANK
1	1	1	1
1	2	1	1
1	3	1	1
1	4	1	1
2	5	5	2
2	6	5	2
3	7	7	3

Table 10.1: Ranking function differences when ordering by the ID columns

```
SELECT
    CORR( CashDiff, EZDiff ) as DiffCor
FROM
    (SELECT
        LAG(vehiclesEZ) OVER(PARTITION BY plaza
                ORDER BY mtadt ASC, hr asc)
            - vehiclesEZ AS EZDiff
        , LAG(vehiclescash) OVER(PARTITION BY plaza
                ORDER BY mtadt ASC, hr asc)
            - vehiclescash AS CashDiff
    FROM
        cls.mta ) as innerQ;


  diffcor
---------
 0.744516
```

- These types of functions are evaluated *after* with SELECT after GROUP, JOIN, WHERE and HAVING. This means that you can't refer to them within those functions. If you want to filter on a window function it must be contained within a subquery.
- An important caveat when using these functions, as said from the documentation (emphasis mine):

  > By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, **plus any following rows that are equal to the current row** according to the ORDER BY clause. When ORDER BY is omitted the default frame consists of all rows in the partition.

  This is weird:

```
select
    plaza, mtadt, hr, direction
    , vehiclesez
    , sum(vehiclesez)
        over(partition by plaza order by mtadt, hr
                rows between unbounded preceding
                        and current row) as runningS1
    , sum(vehiclesez)
        over(partition by plaza order by mtadt, hr ) as runningS2
from
        cls.mta
where plaza = 1 and mtadt = '2010-01-01' and hr < 3;


  plaza  mtadt         hr  direction      vehiclesez    runnings1    runnings2
-------  ----------  ----  -----------  ------------  -----------  -----------
      1  2010-01-01     0  I                     415          415          801
      1  2010-01-01     0  O                     386          801          801
      1  2010-01-01     1  I                     702         1503         2037
      1  2010-01-01     1  O                     534         2037         2037
      1  2010-01-01     2  I                     559         2596         3106
[...]
```

This is weird because when you omit ROWS BETWEEN, the running sum is computed as if rows which have similar values in the partition are the same. The same query however, with a ROWS BETWEEN clause computes a running sum while ignoring the duplicate rows.

2 Using Analytic Functions with Transaction Data

- In this section we return to trying to understand the revenue behavior of our soap transaction data. Just like before we are going to use the notion of a cohort to help our analysis.
- Consider the data in Table 9.1 which contains information on users who were making certain transactions.
- Let's begin by calculating the revenue per user by locale and also by their install time period.
- We didn't even attempt this in the previous section because it involved so many joins! Using Analytic functions allows us to skip many of those issues!

DRAFT

```
select
    cohort
    , locale
    , count(distinct userid) as numusers
    , sum(case when trans_dt::date <= (cohort + '1 month'::interval)::date
        then amt else 0 end ) as mon_0_amt
    , sum(case when trans_dt::date <= (cohort + '2 month'::interval)::date
        then amt else 0 end ) as mon_1_amt
    , sum(case when trans_dt::date <= (cohort + '3 month'::interval)::date
        then amt else 0 end ) as mon_2_amt
from
( select
    first_value(locale) over(partition by userid order by trans_dt asc ) as locale
    , date_trunc('month', first_value(trans_dt)
        over(partition by userid order by trans_dt asc))::date as cohort
    , amt, userid, trans_dt
from
    cls.trans ) as innerQ
GROUP BY 1,2


cohort      locale      numusers    mon_0_amt    mon_1_amt    mon_2_amt
----------  --------  ----------  -----------  -----------  -----------
2016-01-01  Canada          4706       162262       167287       223375
2016-01-01  Mexico          3920       186854       190914       224451
2016-01-01  U.S.           12676       542198       599027       637072
2016-02-01  Canada          4334       147535       152976       206147
2016-02-01  Mexico          3602       171472       175171       208097
[...]
```

- Let's calculate the percentage of revenue that each transaction represents for each userid (how would we do this without Analytic Functions?):

```
select
    userid, trans_dt, amt
    , amt/sum( amt) over(partition by userid) as pct
from
    cls.trans


  userid  trans_dt      amt    pct
--------  ----------  -----  -----
       1  2016-05-09  23.98    1
       2  2018-08-25  12.99    1
       3  2017-03-05  43.16    0.5
       3  2017-04-05  43.16    0.5
       4  2016-02-28  59.95    1
[...]
```

- I want to calculate the percentage likelihood that a person who has made X purchases makes another one. There are a number of different ways that this can be done, but we can use analytic functions:

```
select
    transNum
    , sum( case when transNum = totalTrans then 1 else 0 end)::float
    / count(1) as pct
    , count(1) as numerator
from
(select
    row_number() over(partition by userid order by trans_dt) as transNum
    , count(1) over(partition by userid) as totalTrans
from
    cls.trans) as innerQ
group by 1
order by 1;


  transnum       pct    numerator
----------  --------  -----------
         1  0.529178       574289
         2  0.538423       270388
         3  0.58621        124805
         4  0.644172        51643
         5  0.686276        18376
[...]
```

3 Common Table Expressions ("CTE")

- A relatively new piece of SQL syntax is WITH, which allows for tables to be defined and used repeatedly within a query. These are called Common Table Expressions. CTE are incredibly powerful ways of writing queries, but they can come with significant downsides (as we will discuss later).

  From PostgreSQL's documentation:

  > A useful property of WITH queries is that they are evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to avoid redundant work. Another possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary sub-query. The WITH query will generally be evaluated as written, without suppression of rows that the parent query might discard afterwards. (But, as mentioned above, evaluation might stop early if the reference(s) to the query demand only a limited number of rows.)

- The motivation for CTE is that they can increase readability in query by defining a table at the start of your query which only exists for the duration of the query.
- The WITH clause is used to start a CTE and it basically sets up a derived table that can be used in the query.
- Consider the following example:

DRAFT

```
with only_inbound as (select * from cls.mta where direction = 'I')

select * from only_inbound limit 100;


  plaza  mtadt          hr  direction      vehiclesez    vehiclescash
-------  ----------   ----  -----------  ------------  --------------
      2  2013-10-14     16  I                    2469             336
      2  2013-10-14     17  I                    2853             425
      2  2013-10-14     18  I                    2575             394
      2  2013-10-14     19  I                    2422             344
      2  2013-10-14     20  I                    1989             339
[...]
```

The basic syntax of the query is that we define a table via a query at the start using a `WITH` clause. This table does not have a schema and can only be referenced within that query.

- We can use CTEs with multiple queries by separating them with commas:

```
with
    only_inbound as (select * from cls.mta where direction = 'I')
    , only_outbound as (select * from cls.mta where direction = 'O')

select
    plaza, mtadt, hr
    , only_inbound.vehiclesez as inbound_ez
    , only_outbound.vehiclesez as outbound_ez
 from
    only_inbound
join
    only_outbound
using( plaza, mtadt, hr)

limit 10;


  plaza  mtadt          hr    inbound_ez    outbound_ez
-------  ----------   ----  ------------  -------------
      2  2013-10-14     17          2853           2116
      2  2013-10-13     11          2960           2081
      2  2013-10-12     17          2847           2433
      2  2013-10-10      1           189            177
      2  2013-10-10      3           118            140
[...]
```

- Where I find CTEs to be useful is when there are multiple layers of logic that need to be implemented. By using a CTE I can break up that application logic into separate pieces that are easier to read.

4 CTEs with the transaction data

- To use the WITH clause you specify a table name and then use AS. It is done before the SELECT in the query. For example, the following creates a table that only looks at unit transactions from the United States. We then use this figure out the average order value of these transactions:

```
with USUnits as (
    select * from cls.trans
    where locale = 'U.S.' and type = 'Units')

select avg(amt) as AOV from USUnits;



    aov
-------
43.3045
```

- Consider the following, more useful, example which creates an LTV dataset which has the first value of the local of purchase.

```
with LTVData as (select
    first_value(locale) over(partition by userid order by trans_dt asc ) as locale
    , date_trunc('month', first_value(trans_dt)
        over(partition by userid order by trans_dt asc))::date as cohort
    , amt, userid, trans_dt
from
    cls.trans )

select * from LTVData where locale = 'U.S.' limit 100;


locale    cohort       amt    userid  trans_dt
--------  ----------  -----  --------  ----------
U.S.      2016-05-01  23.98         1  2016-05-09
U.S.      2017-03-01  43.16         3  2017-03-05
U.S.      2017-03-01  43.16         3  2017-04-05
U.S.      2016-02-01  59.95         4  2016-02-28
U.S.      2016-01-01  99.95         6  2016-01-05
[...]
```

- We can also have multiple tables defined:

```
with
    LTVData as (select
        first_value(locale) over(partition by userid order by trans_dt asc ) as locale
        , date_trunc('month', first_value(trans_dt)
            over(partition by userid order by trans_dt asc))::date as cohort
        , amt, userid, trans_dt
    from
        cls.trans )
    , SubScribersFirst as (select distinct userid from
        (select userid, first_value( type ) over(partition by userid
            ORDER BY trans_dt asc, type asc ) as firsttype
            from cls.trans) as innerQ
        where firsttype = 'Sub')
select
    *
from
    SubScribersFirst
left join
    LTVData
using(userid);


  userid  locale    cohort        amt  trans_dt
--------  --------  ----------  -----  ----------
       2  Canada    2018-08-01  12.99  2018-08-25
       3  U.S.      2017-03-01  43.16  2017-03-05
       3  U.S.      2017-03-01  43.16  2017-04-05
       5  Canada    2018-03-01  17.98  2018-03-09
       5  Canada    2018-03-01  17.98  2018-05-09
[...]
```

- The upside of using a CTE is that they can be much easier to read.
- There are two major downsides to using CTEs:
  1. Some databases do not support them (MySQL)
  2. In other databases they can act as optimization barriers. In particular, consider the following query:

DRAFT

```
with
    LTVData as (select
        first_value(locale)
            over(partition by userid order by trans_dt asc ) as locale
        , date_trunc('month', first_value(trans_dt)
            over(partition by userid order by trans_dt asc)) as cohort
        , amt, userid, trans_dt
    from
        cls.trans )
select
    *
from
    LTVData
where userid = 2;


locale    cohort                         amt    userid  trans_dt
--------  -------------------------  -----  --------  ----------
Canada    2018-08-01 00:00:00+00:00  12.99         2  2018-08-25
```

In this example, it is clear that the filter WHERE userid = 2 could be applied within the LTVData expression. However, it is not and the database will compute the entire LTVData before applying the filter, a costly choice. We will explore performance considerations in the next section.

DRAFT

Chapter 11

Database Internals: Performance Evaluation

DRAFT

Contents

1 Normalization 183
2 Views 185
3 Information Schema 189
4 Performance Considerations 190
5 Index 194
6 Distributed Systems and the CAP Theorem 195

DRAFT

1 Normalization

- Up to this point we have taken the data in a database as a given without consideration as to what a database *should* look like. In this section we will introduce some of the major concepts around these database design decisions, starting with the process of normalization.
- To motivate this, we need to go back in time and consider one of the common critiques of the relational system originally proposed by Codd, which was that when data within a database is changed there were possible negative side effects. The database normalization process is a set of rules, which if done correctly, will limit the likelihood of this these issues, called *anomalies* occurring.
  1. **Deletion Anomaly:** Non-database information lost due to deleting data within the database.
  2. **Insertion Anomaly:** Incomplete data may mean that we cannot insert information while keeping the database consistent.
  3. **Update Anomaly:** When the database is updated, multiple updates may be required to maintain consistency.
- In order to understand these issues, consider the following tables describing faculty at a school.

Figure 11.1: *faculty* table

FID	Faculty	HireDt	Dept	Dean1	Dean2	Mentor
1	Hamrick	8/95	Finance	Big Boss	Little Boss	Afraid
2	Ross	2/12	Accounting	Small Boss	Medium Boss	Uminsky
3	Parr	2/85	CS	Medium Boss	Small Boss	Hamrick
4	Uminsky	8/11	Math	Big Boss	Biggest Boss	Tao

- Using the table above, how do we write a query which lists all the departments?

```
select distinct dept from faculty;
```

- Using the table above, how do we write a query which returns the number of faculty?

```
select count(*) from faculty;
```

- What can go wrong:
  - **Deletion Anomaly:** What happens when the last Ancient Greek Professor retires? The query above will no longer return Ancient Greek as a department – but the department may still exist!
  - **Insertion Anomaly:** We decide to introduce a new major, but have not hired a professor in that department yet. The query above will not reflect the new department. If we add a row with Null professor to add this information the database becomes inconsistent because the second query above no longer returns the expected information.
  - **Update Anomaly:** In order to get more students, Mathematics decides to change the name of their department to "Mathematical Sciences." In order to make this single "fact" change *every* row in the database related to a math professor must change. if something happened in the middle of the update or if some professor misreported their major next year we would have two math departments when only one should exist.
- How do we avoid this? We *normalize* the table. There are a bunch of different criteria for normalization forms. In our case we will go over a few of the most common ($1^{st}, 2^{nd} and 3^{rd}$ normal forms.

DRAFT

Wikipedia currently lists six ordinal normal forms as well as a few named other version, the most commonly mention (in my experience), being Boyce-Codd Normal Form ("BCNF").

- First Normal Form:
  1. All values within a cell are *atomic* – there is not a cell which contains multiple values. For example, there is no column phone numbers which contains multiple phone numbers in the same row.
  2. Each table has a *key* which logically defines a record.
  3. No repeating columns.
- In our example we violate the no repeating columns rule! So we rewrite the table to avoid this:

Figure 11.2: *faculty* table

FID	Faculty	HireDt	Dept	Dean	DeanNo	Mentor
1	Hamrick	8/95	Finance	Big Boss	1	Afraid
1	Hamrick	8/95	Finance	Little Boss	2	Afraid
2	Ross	2/12	Accounting	Small Boss	1	Uminsky
2	Ross	2/12	Accounting	Medium Boss	2	Uminsky
3	Parr	2/85	CS	Medium Boss	1	Hamrick
3	Parr	2/85	CS	Small Boss	2	Hamrick
4	Uminsky	8/11	Math	Big Boss	1	Tao
4	Uminsky	8/11	Math	Biggest Boss	2	Tao

In this table, what logically defines a row is the faculty-dean combination.

- While the table above moves us toward avoiding the anomalies mentioned above (for example, it is now way easier to change a Dean's name), it still is not a great solution.
- 2nd normal form continues this process, by adding an additional constraint:
  1. All the rules from first normal form
  2. No secondary key (or subset of other key) can have a functional dependency on another attribute. Generally this occurs when you have repeating rows of data related to the key.

In order to implement this normalization procedure we will need to create a new table since we cannot have any repeating rows.

FID	Faculty	HireDt	Dept	Mentor
1	Hamrick	8/95	Finance	Afraid
2	Ross	2/12	Accounting	Uminsky
3	Parr	2/85	CS	Hamrick
4	Uminsky	8/11	Math	Tao

FID	Dean	DeanNo
1	Big Boss	1
1	Little Boss	2
2	Small Boss	1
2	Medium Boss	2
3	Medium Boss	1
3	Small Boss	2
4	Big Boss	1
4	Biggest Boss	2

Figure 11.3: Second Normal Form

DRAFT

- However, this doesn't alleviate all possibility of all the anomalies that we have listed. In order to do that we can try to put the database into third normal formal:
    1. All the rules for second normal form
    2. "Facts" should be independent of the key information that they contain XXX
- In order to put this database in third normal form we need to add *a lot* of tables.

FID	Faculty	HireDt
1	Hamrick	8/95
2	Ross	2/12
3	Parr	2/85
4	Uminsky	8/11

FID	MentorFID
1	92
2	4
3	1
4	85

FID	DID	DeanNo
1	1	1
1	2	2
2	3	1
2	4	2
3	4	1
3	3	2
4	1	1
4	5	2

DeptID	DeptName
10	Finance
12	Accounting
13	CS
40	Math

DID	Dean
1	Big Boss
2	Little Boss
3	Small Boss
4	Medium Boss
5	Biggest Boss

FID	DeptID
1	10
2	12
3	13
4	40

Figure 11.4: Third Normal Form

DRAFT

- The positive of putting the database into third normal form is that you minimize the risk of creating one of the anomalies specified above.
- There are, however, a few downsides:
    1. Many, many tables were created.
    2. The additional tables will need additional effort by the database admins (cost)
    3. With the addition of some many tables, queries may require multiple joins. Doing additional joins may put additional stress on the database.

2 Views

- Normalization frequently leads to schema with lots and lots of tables and, with that, lots of joins that get repeated over and over again in queries. Luckily, relational databases have a system for storing queries within the database without replicating the data. This structure is called a *view* and can be considered a stored query that gets accessed like a table.

- For example lets consider the case where we frequently want to look at Motor Homes from Polk county in our Iowa cars table. We could write the following query to get the data:

```
select * from cls.cars where countyname = 'Polk' and vehiclecat = 'Motor Home';
```

  but if we were doing this frequently we could store this as a view instead using the following command:

```
CREATE VIEW cls.polk_motor_homes as (select * from cls.cars where countyname = 'Polk' and vehiclecat = 'Motor Home');
```

- Let's take a look at the following query and what it returns:

```
select
        table_name
        , table_schema
        , table_type
from information_schema.tables
where table_name in ('cars', 'columns');

 table_name |    table_schema    | table_type
------------+--------------------+------------
 columns    | information_schema | VIEW
 cars       | cls                | BASE TABLE
```

- The table type for cars is called a "BASE TABLE" and is the standard table type in the database.
- The columns table, on the other hand, is not a table, it is a view. A **view** is a stored query masquerading as a table. If we run the following query and look at the results you can see the underlying query!

```
select
        table_name
        , view_definition
from
        information_schema.views
where table_name = 'columns';

 table_name |                        view_definition
 ------------+-----------------------------------------------------
 columns    |  SELECT (current_database())::information_schema.s
```

  The view_definition has been truncated as the query underlying this view is long. The premise, however, is straightforward: each time the table "columns" is accessed, the query in the view definition is executed.

- We can define a view using any select query. If we are analyzing Motorcycle registrations from Lucas County frequently we could execute the following query:

```
 create view cls.mc_lucas as
        select
                countyname
                , year
                , registrations
        from cls.cars
                where countyname ='Lucas'
                and vehicletype = 'Motorcycle';
```

This would create a view which we could easily access using SQL and it will also be in the information_schema:

```
select *
        from mc_lucas
order by year;

 countyname | year | registrations
 ------------+------+---------------
 Lucas      | 2005 |           530
 Lucas      | 2006 |           586
 Lucas      | 2007 |           606
 Lucas      | 2008 |           592
 Lucas      | 2009 |           587
 Lucas      | 2010 |           588
 Lucas      | 2011 |           578
 Lucas      | 2012 |           582
 Lucas      | 2013 |           586


select
        table_name
        ,table_type
from
        information_schema.tables
where table_name = 'mc_lucas';

 table_name | table_type
------------+------------
 mc_lucas   | VIEW
```

- As before, the query defining the view is found in the "views" table:

```
select
        table_name, view_definition
from information_schema.views
where table_name = 'mc_lucas';

-[ RECORD 1 ]---+------------------------------------------------------
table_name      | mc_lucas
view_definition |  SELECT cars.countyname,                            +
|   cars.year,                                                       +
|   cars.registrations                                               +
|   FROM cls.cars                                                    +
|   WHERE (((cars.countyname)::text = 'Lucas'::text)                 +
|   AND ((cars.vehicletype)::text = 'Motorcycle'::text));
```

As shown above, the original query defining the view has been modified by the database. This modification does not change what the query returns.

- In order to get rid of the view, we use the DROP VIEW command:

```
drop view cls.mc_lucas;
```

At which point in time the view is removed.

- Why are views useful?
  - **Shared Query**: If an entire team is doing a set of analysis on a particular topic, using a view to create a common table can ensure consistency between different team members.
  - **Version Control**: As team members learn about a topic they can use views to keep everyone up-to-date with the latest findings. For example, a table with the name "Fraud" which detects fraudulent transactions can be updated to include new algorithms for detecting fraud without everyone having to re-write their queries.
  - **Laziness:** Using a view means less typing for everyone!
- Downside of views:
  - **Performance considerations:** For a number of reasons, views tend to be less performant than using a real query. In some relational databases, views can act as an *optimization barrier*. Consider the following example:

    ```
    select * from cls.mc_lucas where county = 'Lucas';
    ```

    In this example, the underlying mc_lucas view removes all rows which do not refer to Lucas county. However, in *some* circumstances the view will act as an optimization barrier and check each row twice, once in the view and once in the outer where, to verify that the rows are from Lucas county.
  - **Proliferation of Views:** If everyone on a data team is allowed to create views the tendency is for the number of views to explode. If there are too many views it becomes difficult to find and they go unused.

DRAFT

3 Information Schema

- Returning to Codd and his 4th rule:

  > The database is represented at the logical level the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data

- What does this mean? Reading it closely, note that Codd is saying that *the database* itself is represented the same way as *ordinary data*. Users of the database should be able to use the same relational language (SQL) to get answer questions about the database.
- In other words: Everything about the database is in the database itself.
- Unfortunately, this representation is specific to the SQL variant. Postgres contains database specific information in a variety of places, though the most common place to access it is via the tables in the schema "information_schema"
  - Tables contains information on the tables within the database.
  - Columns contains column specific information
  - Views contains information on views, which we will discuss later.
- For example:

```
select
        *
from
        information_schema.columns
where
        column_name = 'registrations';
```

  Will return all the information about the registrations column from the cars table.

- Detailed information about the columns in each table:

DRAFT

```
select
        column_name
        , data_type
from
        information_schema.columns
where table_name ='cars';

column_name    |     data_type
-------------------+-------------------
year              | integer
yearending        | date
countyname        | character varying
countycode        | integer
motorvehicle      | character varying
vehiclecat        | character varying
vehicletype       | character varying
tonnage           | character varying
registrations     | integer
annualfee         | double precision
primarycountylat  | double precision
primarycountylong | double precision
primarycountycord | character varying
```

With this query we can identify the data types associated with each column, which should match the data types used to define the table.

- We can write queries against the information schema using any SQL functionality. For example, we could identify all the character columns with the following:

```
select *
from
        information_schema.columns
where
        data_type like 'char%';
```

- A common question asked by new users of SQL is if it is possible to store additional data about the database within a table. The answer is "Yes" – you can create a table which contains information about the database, but that table will have to updated by hand. There is no automatically generated data dictionary created when using a relational database.

4 Performance Considerations

Before talking about query performance and how to measure it we need to discuss two important questions:

1. How the data is stored on the hard drive:

    - The easiest mental model for how data is stored on a hard drive is to consider the database as a record player, where each row is stored on the hard drive in a random order, back-to-back. In other words, where one row ends another row begins.

- There are a couple of interesting aspects to why this is important. First off, if every row is the same length then finding the start and end of a row can be pretty easy. For example, if every row is 25 bytes long, then going to the fifth row entails simply seeking ahead $25 \cdot 5 = 125$ bytes. If the rows are arbitrarily long, then this process can take some time.
- The above is why there is a difference between char and varchar data types. A char datatype is a *fixed* length while a varchar is of variable length. In some database systems storing data as a char can increase performance significantly.[1]
- In order to find data on a database, the computer must seek around the drive, which is incredibly slow, even on modern databases. Consider the following table which describes the relative speed of different random access levels. [2]

Figure 11.5: Access Speeds

Access type	Actual time	Approximated time
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

  Looking at this table, you can see that seeking data, even on an SSD, can be incredibly slow in comparison to processing speeds. Fundamentally, a database stores data and attempts to provide access to it as quickly as possibly – so it is always running up against this hurdle.

2. How the database decides how to execute a particular query:
   - Basically, the database takes your SQL query and runs it through a database optimizer which rewrites your query a number of different ways and then estimates how long each way will take. The database will then execute the one that it believes will take the smallest amount of time.
   - The query optimizer can often be wrong.
   - There are a number of different ways that it can be wrong, but most of them relate to "cost" – or how long the database thinks a particular operation will cost. In order to estimate the cost, the database keep statistics on each table and column. For example, consider the following:

[1] This is not true of Postgres, which stores both varchar and char the same way.
[2] Source: `https://madusudanan.com/blog/understanding-postgres-caching-in-depth/`

DRAFT

```
 select * from pg_stats where tablename = 'mta' and attname = 'direction' ;
-[ RECORD 1 ]----------+---------------
schemaname             | cls
tablename              | mta
attname                | direction
inherited              | f
null_frac              | 0
avg_width              | 2
n_distinct             | 2
most_common_vals       | {I,O}
most_common_freqs      | {0.5276,0.4724}
histogram_bounds       |
correlation            | 0.450035
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

This query contains information on the direction column within the MTA table. You can see that there is a host of information about the contents. This includes the frequency information. However:

```
 select count(1), direction from cls.mta group by 2;
-[ RECORD 1 ]-----
count     | 613608
direction | I
-[ RECORD 2 ]-----
count     | 552120
direction | O

ncross=# select 613608.0 / ( 552120 + 613608.0);
-[ RECORD 1 ]--------------------
?column? | 0.52637321913859836943
```

The frequency information is not correct!

- The database itself does not keep perfect statistics on each column, it would take too long and instead runs on estimates that it will occasionally update.

- So how can we know what the database is doing and if it is being efficient?
- Every major relational database system has a query analyzer or "explain" which gives you insight into what the database is doing.
- With Postgres, there are two commands that are used to do this: EXPLAIN and EXPLAIN ANALYZE. Generally speaking these commands do not show up nicely when using an SQL client, using psql will result in better output.
- EXPLAIN will tell you what the query optimizer *thinks* will happen while EXPLAIN ANALYZE will return both what the query optimizer thought would happen and what actually happened.
- Let's look at the following example:

```
ncross=# explain analyze select plaza, sum(hr) from cls.mta where hr = 1 group by 1 having plaza = 2;
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=0.00..24937.72 rows=1 width=8) (actual time=149.688..149.688 rows=1 loops=1)
   Group Key: plaza
   ->  Seq Scan on mta  (cost=0.00..24911.92 rows=5159 width=8) (actual time=15.469..148.941 rows=5124 loops=1)
         Filter: ((hr = 1) AND (plaza = 2))
         Rows Removed by Filter: 1160604
 Planning time: 0.074 ms
 Execution time: 149.731 ms
(7 rows)

ncross=# explain analyze select plaza, sum(hr) from cls.mta where plaza = 2 and hr = 1 group by 1;
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=0.00..24937.72 rows=1 width=8) (actual time=137.129..137.129 rows=1 loops=1)
   Group Key: plaza
   ->  Seq Scan on mta  (cost=0.00..24911.92 rows=5159 width=8) (actual time=13.293..136.394 rows=5124 loops=1)
         Filter: ((plaza = 2) AND (hr = 1))
         Rows Removed by Filter: 1160604
 Planning time: 0.071 ms
 Execution time: 137.163 ms
(7 rows)
```

- In the above we can see that the query plan is *the same* for the two queries despite the use of the HAVING clause in the first. In other words, the database was smart enough to remove the plazas which were not equal to 2 before doing the aggregation.
- In order to read these you start from the bottom and go up.
- Another one. CTE acting as optimization blocker

```
explain analyze with
        MTALTV as (select
            plaza, mtadt, hr, direction
            , sum(vehiclesez) over(partition by plaza order by mtadt, hr rows between unbounded preceding and current row) as runningS
        from
                cls.mta)
select
        *
from
        MTALTV
where plaza =1 and hr = 0 and mtadt = '2010-01-01' and direction = 'I'

                                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 CTE Scan on mtaltv  (cost=186684.07..221655.91 rows=1 width=28) (actual time=898.385..1960.692 rows=1 loops=1)
   Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date) AND ((direction)::text = 'I'::text))
   Rows Removed by Filter: 1165727
   CTE mtaltv
     ->  WindowAgg  (cost=160455.19..186684.07 rows=1165728 width=22) (actual time=898.361..1541.795 rows=1165728 loops=1)
           ->  Sort  (cost=160455.19..163369.51 rows=1165728 width=18) (actual time=897.422..1058.579 rows=1165728 loops=1)
                 Sort Key: mta.plaza, mta.mtadt, mta.hr
                 Sort Method: external merge  Disk: 34232kB
                 ->  Seq Scan on mta  (cost=0.00..19083.28 rows=1165728 width=18) (actual time=0.022..202.735 rows=1165728 loops=1)
 Planning Time: 0.293 ms
 Execution Time: 1985.602 ms
(11 rows)
```

```
explain analyze with
        MTALTV as (select
            plaza, mtadt, hr, direction
            , sum(vehiclesez) over(partition by plaza
                    order by mtadt, hr rows
                        between unbounded preceding and current row) as runningS
        from
                cls.mta
                where plaza =1 and hr = 0 and mtadt = '2010-01-01' and direction = 'I')
select
        *
from
        MTALTV
limit 100;

                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Limit  (cost=27826.27..27826.31 rows=2 width=20) (actual time=19.346..147.565 rows=2 loops=1)
   CTE mtaltv
     ->  WindowAgg  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.342..147.555 rows=2 loops=1)
           ->  Seq Scan on mta  (cost=0.00..27826.24 rows=2 width=16) (actual time=19.314..147.520 rows=2 loops=1)
                 Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date))
                 Rows Removed by Filter: 1165726
   ->  CTE Scan on mtaltv  (cost=0.00..0.04 rows=2 width=20) (actual time=19.344..147.561 rows=2 loops=1)
 Planning time: 0.108 ms
 Execution time: 147.609 ms
(9 rows)
```

```
 explain analyze select
        plaza, mtadt, hr
        , sum(vehiclesez) over(partition by plaza
            order by mtadt, hr rows
            between unbounded preceding and current row) as runningS
    from
        cls.mta
        where plaza =1 and hr = 0 and mtadt = '2010-01-01' limit 100 ;
                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.450..137.015 rows=2 loops=1)
   ->  WindowAgg  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.448..137.011 rows=2 loops=1)
         ->  Seq Scan on mta  (cost=0.00..27826.24 rows=2 width=16) (actual time=19.435..136.992 rows=2 loops=1)
               Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date))
               Rows Removed by Filter: 1165726
 Planning time: 0.116 ms
 Execution time: 137.065 ms
```

- These commands give us a ton of insight into how a query is operating.

5 Index

- So what do we do if a query is slow?
- The easy answer is usually to add an *index*, which is a tree structure which allows for searching for values quickly. Specifically, the goal of an index is to decrease the number of *random* hard drive accesses.

```
explain analyze select retdate, symb, sum(cls) over(partition by symb order by retdate asc) from stocks.s2010 limit 10;

                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Limit  (cost=81387.26..81387.46 rows=10 width=40) (actual time=14700.353..14700.428 rows=10 loops=1)
   ->  WindowAgg  (cost=81387.26..92327.78 rows=547026 width=40) (actual time=14700.351..14700.424 rows=10 loops=1)
         ->  Sort  (cost=81387.26..82754.82 rows=547026 width=40) (actual time=14699.988..14700.059 rows=11 loops=1)
               Sort Key: symb, retdate
               Sort Method: external merge  Disk: 23720kB
               ->  Seq Scan on s2010  (cost=0.00..14293.26 rows=547026 width=40) (actual time=0.038..481.515 rows=816066 loops=1)
 Planning time: 1.862 ms
 Execution time: 14715.666 ms
(8 rows)
```

vs.

```
create index tst2 on stocks.s2010 (symb, retdate);

 explain analyze select retdate, symb, sum(cls) over(partition by symb order by retdate asc) from stocks.s2010 limit 10;
                                                           QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.42..1.31 rows=10 width=16) (actual time=0.388..1.308 rows=10 loops=1)
   ->  WindowAgg  (cost=0.42..72501.81 rows=816066 width=16) (actual time=0.386..1.302 rows=10 loops=1)
         ->  Index Scan using tst2 on s2010  (cost=0.42..58220.66 rows=816066 width=16) (actual time=0.301..1.077 rows=11 loops=1)
 Planning time: 2.509 ms
 Execution time: 1.715 ms
(5 rows)

drop index stocks.tst2;
```

- So why not use Index's everywhere?
- The downside of using index is that they take additional hard drive space and increase the cost of any additional data writes or updates. Since the index has to be kept up to date with the data on disk this means that all writes or updates to any table needs to be reflected in changes to all appropriate indexes.
- There are many different types of indexes and these different types allow for efficiencies for different access methods. Examples include things like B-trees which allow for fast comparison operators

6 Distributed Systems and the CAP Theorem

- In this section we will start talking about *distributed* systems, or multiple computer systems which are networked together to act as a single unit.
- When dealing with these systems, each computer is generally refereed to as a "node" and the entire system called a "cluster."
- Distributed systems work by spreading work around different nodes. For example, if you have a 10-node cluster and send it two queries, those queries will not necessarily land on the same node each time and thus the physical machine returning the result to you might be different.
- Many instances require distributed systems because the amount of data generated is too large to be handled by a single computer.
- Consider the following examples:
  1. **Zynga (2014)**

     500-1000 Node Vertica cluster

     60B rows/day

     10 TB/day
  2. **Sega (2015)**

     8 Node Redshift cluster

     20 TB

     Main Table had 4B rows

     110MM rows/day

     Incredibly spiky data flows
  3. **GSN (2014)**

     200 Node Vertica cluster

DRAFT

Every 15 minutes 5MM rows loaded

Biggest table had 100B rows

- An important mathematical theorem governing these types of systems is called the CAP theorem, which is a non-existence result. The CAP theorem states that no distributed system can be **C**onsistent, **A**vailable and **P**artition tolerant. You can only choose two of these three things. Note that this theorem is a mathematical result and like many such results there is an ongoing discussion of how well the mathematical definitions of these terms line-up with reality. That discussion is beyond the scope of this class.
  - **Consistency:** A system is consistent if all nodes within the system respond the same way to the same query. For example, if you send a query to the first node in a cluster and it responds with data X then that same query, if the second node responded, should also return data X.[3]
  - **Availability:** A system is available if, when one node goes down, the system is still able to respond to questions.
  - **Partition Tolerance:** Nodes will continue to perform even if there is a disruption in communication between them.
- As an example, consider the following story of a startup that you create call 1-800-REMINDME:[4]
  - The basic business model is that people call 1-800-REMINDME and then state their name and something that they want to remember, such as "Jeff, I have a meeting at 8AM."
  - If you call back it will return what you said the first time and charge them 10 cents.
  - **Day #1:** You sit down with your notebook and start writing down and responding to calls. This works great!
  - **Day #2:** TechCrunch and VentureBeat both publish articles about your hot new startup. You are now swamped and when people call they are getting busy signals!
  - **Day #3:** You add your sig-o to help, who sits in another room, on another line, with her own notebook. Unfortunately, this doesn't work! Sometimes people call and get your sig-o, sometimes they call and get you, but either way, some messages end up in their notebook and some in yours:

```
Jeff: When is my meeting?
sig-o: You don't have a meeting.
Jeff: Wow. You suck.
```

  - This system is not **consistent**, as, depending on what node you contact you get different responses.
  - **Day #4:** New plan! In order to make the system consistent you do the following: before making any create or update operation confirm it with the other person:

```
Jeff to sig-o: My class is at 8AM
sig-o to you: Jeff, class at 8AM
you to sig-o: Jeff, class at 8AM confirmed.
sig-o to Jeff: confirmed.
```

[3]Note that this is a different type of consistency then the type of consistency we spoke of when talking about transactions and ACID.

[4]This was taken from http://ksat.me/a-plain-english-introduction-to-cap-theorem/

- This slows down the system on writes, but reads are very quick and consistent since both you and your sig-o's notebook are the same.
- **Day #5:** Oh, no, sig-o get sick and sometimes run out of the room to throw-up! What happens then...

```
Jeff to you: My meeting is at 9AM.
you to sig-o: Jeff, meeting at 9AM.
sig-o: ....
Jeff hangs up after 5 minutes
```

- This system is not **available**, when one node goes down the entire system fails.
- **Day #6:** New idea: If the other person is not reachable then write down all writes and send via email to the other person. The other person does not start answering the phone after coming back before checking their email for a list of all changes and verifying that all those writes are in their system.
- This system is now consistent and available!
- **Day #7:** Your sig-o is pretty sick of this business and just stops talking to you, or sending emails or responding at all. Once again you are stuck, because you don't get an email from them you don't know what changes happened. Because you can't confirm any transactions you can't do any updates or add new clients to your notebook.
- This system is not partition tolerant. If there is no communication between the nodes then the system once again fails. Even though there is perfectly good information for many of 1-800-REMINDME in your notebook, you can't use it because the system that you have in place requires verification from the non-responsive sig-o.

- The CAP theorem states that you can only really choose 2 of the 3 CAP elements.
- Different applications will require different attributes. For example, bank account information will always want consistency – account balances should always report the same number.
- Note that there are also minor definitional changes in each of these. For example, many systems have what is called *eventual* consistency. Redshift, one of Amazon's offerings, promises consistency within 60 seconds.

DRAFT

DRAFT

Chapter 12

Extensions [TBD]

DRAFT

Contents

1 More Advanced Joins **204**
2 OLAP: Cube and Rollup **210**
3 Schemas **210**
4 Keys **210**
5 Data Exploration Strategies **210**
6 Query Strategies **210**

DRAFT

- When we model the LTV of a customer, we usually only consider factors that we can attribute to them at the start of their first session and not factors that occur sometime after their initial start. Why do we do this? Because the numbers that are generated not at the start of a session are hard to compare against. For example, if you take the "LTV of customers who have spent more than than 20 minutes on our website" it is difficult to understand what to compare this to or even what it means.
- The average revenue per customer number that we came up with before is generally not considered to be the LTV. The LTV is usually a cohort measure – it has a time component to it. In the above queries we are comparing users who have spent a short amount of time in the system to those who have spent a lot of time in the system.
- Because LTV is a cohort number, we need to measure the LTV based on the number of users within a time period, such as week or month. For example, lets consider the case where we want to compute LTV based on month of first purchase.

```
select
        , lhs.cohort
        , sum( amt ) as totaldol
        , count(distinct lhs.userid) as numusers
        , sum( amt )/count(distinct lhs.userid) as numusers
from
        (select
                userid, date_trunc( 'month', min(dt ))::date as cohort
        from
                trans
        group by 1) as lhs
left join
        trans
using(userid)
group by 1
```

- The query above only returns the total revenue by cohort. We may want to return the running total, by month. In other words, we want to make a table where the Y-axis is the cohort and the X-axis is the number of months, starting at zero, and the amount of money that was generated by that cohort that many months afterward. Let's consider the first three months of LTV.

  Before starting this, make a chart of what the data should look like:

cohortdt	numusers	mon_0_amt	mon_1_amt	mon_2_amt
01-01-2011	1,155	25,764	12,885	9,995
02-01-2011	355	7,555	3,456	1,111
03-01-2011	755	2,888	7,925	1,100

DRAFT

```
select
        lhs.cohort
        , count(distinct lhs.user_id) as numusers
        , sum(case when trans.dt::date
                between cohort and (cohort + '1 month'::interval)::date
                then amt else 0 end ) as mon_0_amt
        , sum(case when trans.dt::date
                between (cohort + '1 month'::interval)::date
                        and (cohort + '2 month'::interval)::date
                then amt else 0 end ) as mon_1_amt
        , sum(case when trans.dt::date
                between (cohort + '2 month'::interval)::date
                        and (cohort + '3 month'::interval)::date
                then amt else 0 end ) as mon_2_amt
from
        (select
                userid, date_trunc( 'month', min(dt ))::date as cohort
        from
                trans
        group by 1) as lhs
left join
        trans
using(userid)
group by 1
```

- Calculate the average revenue per user in the cohort for the first three months of LTV.

```
select
        lhs.cohort
        , sum(case when trans.dt::date
                between cohort and (cohort + '1 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_0_PU
        , sum(case when trans.dt::date
                between (cohort + '1 month'::interval)::date
                        and (cohort + '2 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_1_PU
        , sum(case when trans.dt::date
                between (cohort + '2 month'::interval)::date
                        and (cohort + '3 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_2_PU
from
        (select
                userid, date_trunc( 'month', min(dt ))::date as cohort
        from
                trans
        group by 1) as lhs
left join
        trans
using(userid)
group by 1
```

- Calculate the average revenue per *active-user* from that cohort for the first two months of LTV:

```
select
        lhs.cohort
        , sum(case when trans.dt::date
                between cohort and (cohort + '1 month'::interval)::date
                then amt else 0 end )
        / count( distinct case when trans.dt::date
                between cohort and (cohort + '1 month'::interval)::date
                then lhs.userid else null end ) as mon_0_PU
        , sum(case when trans.dt::date
                between (cohort + '1 month'::interval)::date
                        and (cohort + '2 month'::interval)::date
                then amt else 0 end )
        / count( distinct case when trans.dt::date
                between (cohort + '1 month'::interval)::date
                        and (cohort + '2 month'::interval)::date
                then lhs.userid else null end ) as mon_1_PU
        , sum(case when trans.dt::date
                between (cohort + '2 month'::interval)::date
                        and (cohort + '3 month'::interval)::date
                then amt else 0 end )
        / count( distinct case when trans.dt::date
                between (cohort + '2 month'::interval)::date
                        and (cohort + '3 month'::interval)::date
                then lhs.userid else null end ) as mon_2_PU
from
        (select
                userid, date_trunc( 'month', min(dt ))::date as cohort
        from
                trans
        group by 1) as lhs
left join
        trans
using(userid)
group by 1
```

- Calculate the LTV for the first three months, per-cohort, for both first-purchase subscribers and first-purchase non-subscribers.

```
select
        date_part('month', firstdt) as cohort
        , subscriber_flag
        , sum(case when trans.dt::date
                between cohort and (cohort + '1 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_0_PU
        , sum(case when trans.dt::date
                between (cohort + '1 month'::interval)::date
                        and (cohort + '2 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_1_PU
        , sum(case when trans.dt::date
                between (cohort + '2 month'::interval)::date
                        and (cohort + '3 month'::interval)::date
                then amt else 0 end ) / count(distinct lhs.user_id) as mon_2_PU
FROM
        (select
                lhs.userid
                , max( case when trans.transtype = 'S' then 1
                        else 0 end ) as subscriber_flag
                , max(firstdt) as firstdt
        from
                (select
                        userid, min(dt) as firstdt
                from
                        trans
                group by 1) as lhs
        left join
                trans
        on
                lhs.userid = trans.userid
                and lhs.firstdt = trans.dt
        group by 1) as lhs

LEFT JOIN
        trans
using( userid)
group by 1,2
order by 2,1;
```

1 More Advanced Joins

In this section we are going to cover a number of common advanced joins.

- In this section we are going to cover some advanced join syntax.

```
        pct          | hr | plaza
---------------------+----+-------
  0.0137942721572615 |  0 |     1
 0.00820199966107439 |  1 |     1
 0.00871038806981867 |  2 |     1
  0.0100321979325538 |  3 |     1
  0.0189798339264531 |  4 |     1
   0.050160989662769 |  5 |     1
  0.0880528723945094 |  6 |     1
  0.0690730384680563 |  7 |     1
  0.0707676664972039 |  8 |     1
  0.0570411794611083 |  9 |     1
  0.0679545839688188 | 10 |     1
  0.0647347907134384 | 11 |     1
  0.0596509066259956 | 12 |     1
  0.0620233858668022 | 13 |     1
  0.0705643111337061 | 14 |     1
  0.0771733604473818 | 15 |     1
  0.0696153194373835 | 16 |     1
  0.0667005592272496 | 17 |     1
  0.0617183528215557 | 18 |     1
  0.0575834604304355 | 19 |     1
  0.0412133536688697 | 20 |     1
  0.0297576681918319 | 21 |     1
  0.0268090154211151 | 22 |     1
   0.023351974241654 | 23 |     1
(24 rows)
```

We join on plaza and then order by hr and this returns the percentage for each hour of the total.

- If we want to do same thing for multiple plazas and multiple days, we can modify the query by removing things from the WHERE clause and adding things to the JOIN:

```
select
        vehiclesez::float / totalez as pct
        , hr
        , lhs.plaza
from
        (select
                vehiclesez
                , mtadt, plaza, hr
        from
                cls.mta
        where
                direction = 'O') as lhs
LEFT JOIN
        (select
                sum( vehiclesez ) as totalez
                , mtadt, plaza
        from
                cls.mta
        where
                direction = 'O'
        group by 2,3) as rhs
using( plaza, mtadt)
order by hr asc;
```

Which should work, but it doesn't because of a division by zero error. How do we handle this? In this case we are going to treat them as NULL:

DRAFT

```
select
        case
                when totalez = 0 then null
                else vehiclesez::float / totalez
                end as pct
        , hr
        , lhs.plaza
from
        (select
                vehiclesez
                , mtadt, plaza, hr
        from
                cls.mta
        where
                direction = 'O') as lhs
LEFT JOIN
        (select
                sum( vehiclesez ) as totalez
                , mtadt, plaza
        from
                cls.mta
        where
                direction = 'O'
        group by 2,3) as rhs
using( plaza, mtadt)
order by plaza, hr;
```

- Let's create a running average for the last two hours (3 data points) of inbound traffic using the EZ-pass for each plaza.

DRAFT

```
select
        lhs.hr, lhs.mtadt, lhs.plaza, lhs.vehiclesez
        , sum(rhs.vehiclesez)::float/count(rhs.vehiclesez) as running
from
        (select
                vehiclesez, hr, mtadt, plaza
        from cls.mta
                where direction = 'I' ) as lhs
left join
        (select
                vehiclesez, hr, mtadt, plaza
        from cls.mta
                where direction = 'I' ) as rhs
on
        lhs.plaza = rhs.plaza
        and (
                (lhs.hr >= 2
                        and lhs.mtadt = rhs.mtadt
                        and rhs.hr >= lhs.hr - 2
                        and rhs.hr <= lhs.hr)
                OR
                (lhs.hr = 1
                        and (
                                (lhs.mtadt = rhs.mtadt and rhs.hr <= 1)
                                or (lhs.mtadt -1 = rhs.mtadt and rhs.hr = 23) )
                )
                OR
                (lhs.hr = 0
                        and (
                                ( lhs.mtadt -1 = rhs.mtadt and rhs.hr in (22, 23))
                                or (lhs.mtadt = rhs.mtadt and rhs.hr=0)
                                )
                )
                )
group by 1,2,3,4;
```

DL

- Keep in mind that the hour goes from 0-23, there is no hour 24 in the dataset. The above query works by matching all rows that fulfill one of three criteria.

  1. If the hour ≥ 2 then just match based on the same date and the right hand side hour is in the previous two hours.
  2. If the hour is equal to 1 then match either the zero or 1 hour with the same date or the 23 hour from yesterday.
  3. If the hour is equal to 0 then match either the hour zero on the same day or hour 22 or 23 from yesterday.

  Once the match is completed the dataset will be three times as large as expected since each row in the left hand table matches *three* in the right hand table. In order to create the average we then use the GROUP BY to collapse those three rows into one, as defined by the left hand table.

- We can also rewrite the query to leverage the contents – and in particular the time aspect of the information. But to do so we will need to convert our hour and time information into a timestamp. There are a number of ways to construct a timestamp, what we will do is combine the date and time using string concatenation and then convert it via the double-colon operator. The following query successfully demonstrates how this would occur:

```
select ('2015-10-04' || ' ' || 2::varchar || ':00')::timestamp ;
      timestamp
---------------------
 2015-10-04 02:00:00
(1 row)
```

- To do the time comparison we will use an INTERVAL operator to assist in the time comparison, as can be seen in the final query below.

```
select
        lhs.mtatime, lhs.plaza, lhs.vehiclesez
        , sum(rhs.vehiclesez)::float/count(rhs.vehiclesez) as running
from
    ( select
            (mtadt || ' ' || hr::varchar || ':00')::timestamp as mtatime
            , plaza, vehiclesez
    from
            cls.mta where direction = 'I' and plaza = 1 ) as lhs
left join
    ( select
            (mtadt || ' ' || hr::varchar || ':00')::timestamp as mtatime
        , plaza, vehiclesez
    from cls.mta where direction = 'I' and plaza = 1 ) as rhs
on
        lhs.plaza = rhs.plaza
        and rhs.mtatime >= lhs.mtatime - interval '2 hour'
        and rhs.mtatime <= lhs.mtatime
group by 1,2,3
order by 1;
```

As in the previous version of this query, we first expand the dataset threefold using the time matching. The data is then collapsed back down into its original form.

- One number we want to calculate is the average revenue per user (ARPU) for a -AAS company. In these situations, customer transaction data is provided in a "long" format with each row representing a single sale.

- Calculating the ARPU is often the first step in calculating a user's lifetime value (or long-term value) ("LTV"). For companies providing a service over time, the calculation of LTV is critical for ensuring profitability. Studying the per-unit economics of a product are often the domain of the data science team and the exercise undertaken below is a common exercise.

- Our major goal for this module is to create a cohort based estimate of how much revenue is generated by an average customer over time. In particular, companies often focus on understanding things like the 90-day ARPU or 120-day ARPU, which represents how much money an average customer generates over the first 90 and 120 days of a customer's lifetime.

- We call this a "cohort" based estimate because we group users based we create this estimate based on when a user first enters the system or alternatively, first becomes a customer. This is usually done on a monthly or weekly basis. So we often consider all new customers who

2 OLAP: Cube and Rollup

TBD

3 Schemas

TBD

Star and Snowflakes

Facts and Dimensions

4 Keys

TBD

5 Data Exploration Strategies

TBD

Goal is to look at data and try to piece how it goes together. Start with just a list of tables and go through "how do we explore this data?"

1. Look at the data (top-10 for each)
2. Think about how it might go together (draw a schema diagram)
3. What do we expect to be unique?
4. How do we test to see if it is unique?
5. What are some one-off things that we can do?

6 Query Strategies

Query strategies: Add a section "Query Strategy" to each lecture which covers some of the more descriptive aspects covered in class.

- Write out what you want.
- Write out conditions for getting it and where data is.
- Build from the inside out.
- For aggregation queries, explain what they doing and dig hard into the data shape.

DRAFT

Chapter 13

Interview Hints

DRAFT

1 Interview Hints

In this section we will go over some advice for preparing for SQL interview questions. The end of this chapter contains a number of real interview questions that candidates have gotten when interviewing over the last few years.

One quick note before jumping into the below. Most of my experience and the experience of students that I speak with is industry focused and specifically “tech” jobs. If you are looking at a position in government, non-profit, or even non-tech fields, the information below may not be as applicable. YMMV.

The three most common ways that SQL is assessed during an interview are:

1. **Recruiter oral assessment**
2. **Whiteboard style assessment**
3. **Automated Assessment**

Before doing interview prep in earnest it is important to think about each interview assessment mechanism and how to best prepare for it.

In the first case, a *recruiter* asking a few SQL questions, the purpose is to filter out people who are bluffing their knowledge. In these cases, the interviewer will ask a question or two based off a simple table that they describe. The person being interviewed is expected to respond with something “that sounds about right”. The recruiter (generally) isn’t writing your code down, they are listening to what you say and comparing it to the solutions that have been given to them.

A whiteboard style assessment is a human evaluated interview style where the interviewer provides the person being interviewed with a series of questions and usually a description of some data structure. While the specifics of this can change a bit the key feature is that you are expected to know the topic before showing up to the interview. Generally speaking some type googling is allowed, but outside of minor syntax (e.g. remembering the order of arguments) it is generally frowned upon. Pre-covid these were all done “on a whiteboard” in an office, but post-covid many of these interviews take place over zoom. Instead of a physical whiteboard, the candidate is expected to share their screen and talk through the problem. You are generally expected to ask questions about the data.

The final common interview tool is an online automated assessment, such as those found on leetcode or hackerrank, though there are a variety of these platforms. In this style of assessment you are provided a prompt (and usually a timer) and sometimes some type of recording / browser lock which prevents you from accessing the internet. You then enter your answers into the online test and submit. Depending on the parameters you may get multiple chances to submit different answers, but most of the time you do not see the response.

So, given the above, how do you prepare?

Recruiter oral assessment

These interviews are easy, the goal is to speak with confidence and get “roughly” the correct answer. The failure mode on this first type of interview is demonstrating a lack of conviction about your knowledge. This is less about what you know and more about knowing a little bit and being convincing.

Whiteboard Interviews

The basic idea is to do lots and lots of problems. In a whiteboard interview you are assessed based on your knowledge, speed and expertise. Having the information at your finger tips is the best way to succeed in these situations. On that note, you’ll find some helpful hints on how to review below.

DRAFT

To prepare for this type of interview you need to *stop writing queries at your computer and start writing queries on paper*. This bears repeating: To prepare for whiteboard SQL questions you need to stop writing queries on your computer, it is a crutch that you will not have during an interview.

I recommend printing out the homework assignments, especially the first few, and answering them on pen and paper. This will train you to stop relying on assistance from the computer (no more auto-complete or syntax highlighting) and focus your energy on what you do not know full understand. In the next section I describe a few "levels" of difficulty that candidates frequently encounter.

Automated Assessment

Automated candidate assessment mechanisms are becoming more and more common in technology related fields. As an interviewer, I (personally) find them to be higher variance than traditional whiteboard interviews. However, in situations where the number of candidates for a position is incredibly high relative to the team size I have been tempted to use these.

That being said, if I were a candidate and faced one of these I would do my best, but accept that it isn't a reflection

DRAFT

There are a few ways that SQL interviews are completed. Even pre-pandemic most SQL interviews were done either over the phone (very simple questions), over a video call or via automated systems (like hacker rank or leetcode). There were some traditional white board interviews with SQL, but that was relatively uncommon. Before jumping into the interviews it is very, very important to keep in mind that SQL, while a "standard" is loose and interviewers will often restrict the available features that are available to the person being interviewed. For example, depending on what options are chosen in the automated system, CTEs and analytics functions may not be available.

In Figure 13.1 you can see an example of a former student who got lucky – CTEs were not allowed in their interview but they were allowed through to the next round. In a bit of foreshadowing, the next round of their interview process ended up doing some whiteboard SQL work and they were cut at that stage.

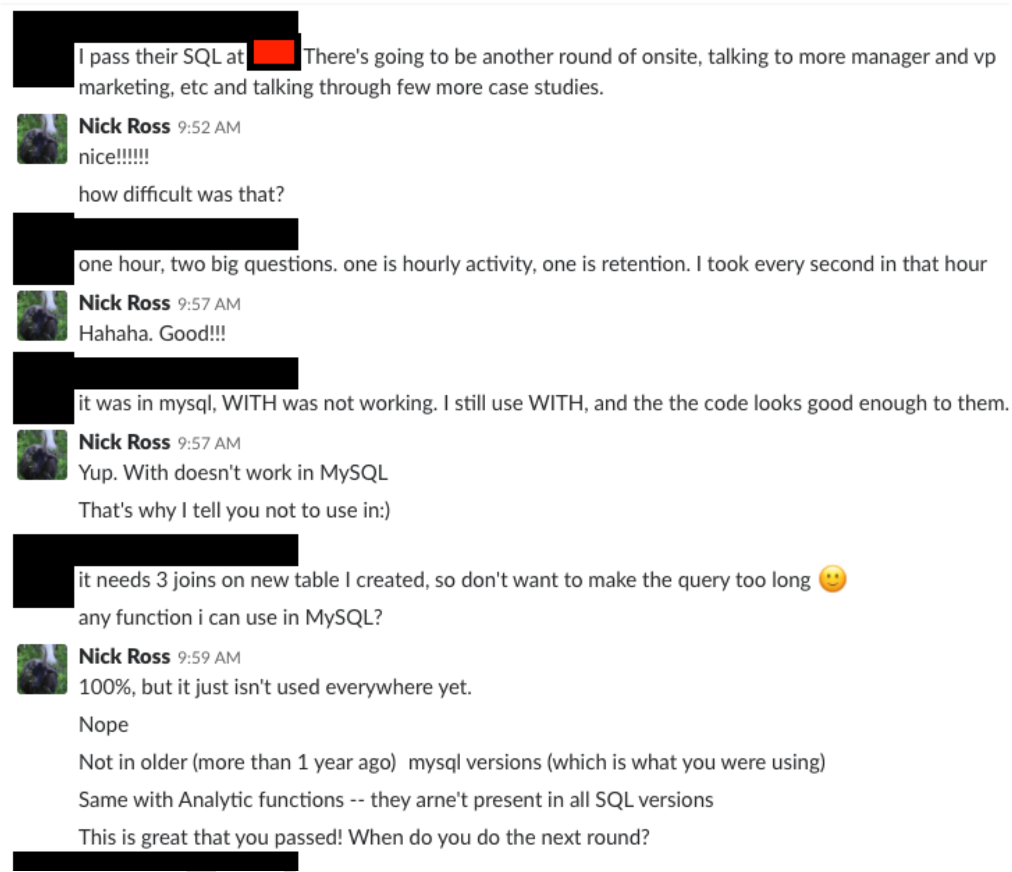


Figure 13.1: Conversation with Student

- **Simple WHERE:** Most questions like this include presenting a table and asking for a query which selects a subset of rows and columns. For example, using the stocks2016 data: write a query which returns all the rows where the volume traded is greater than 10,000. Most of the time, interviewers will not ask specific questions about functions (such as date functions), they are only looking to see if you can write a basic query.

  To prepare for this level of questions:

  - Review the WHERE clause and make sure that you are comfortable answering questions from homework #1. Make sure that you are comfortable writing queries which use SELECT, FROM and WHERE.
  - Focus on making sure you understand basic syntax and how to combine multiple conditions within a WHERE statement using AND and OR.

- Make sure you understand ORDER BY for sorting data.

- **Aggregations:** Questions like this include computing the MAX, MIN, SUM, AVG and COUNT. For example: Count how many rows in stocks2016.d2010 have volume more than 10,000. The purpose of these questions are to make sure that you understand how to collapse data using a GROUP BY. They may also be testing that you understand that WHERE is evaluated *before* the GROUP BY.

  To prepare for this level of questions:

  - Review homework #2. Focus on the syntax and making sure you understand how WHERE and GROUP BY are applied.
  - Understand how to name columns.

- **Complex aggregations with sub queries:** Questions like this include using aggregations on a single table, but multiple times. For example, how many stocks in 2010 have an average price over $50. To complete this you will need to compute the average price for all stocks and then exclude those with an average less than $50 and then count those remaining stocks.

  To prepare for this level of questions:

  - Review homework #3.
  - Focus on naming and how multiple queries fit together.

- **Joins:** Usually this involves either a LEFT or INNER join. The interviewer is trying to test to see if you know the join syntax.

  To prepare for this level of questions:

  - Review Homework #4 as well as read Lesson #4 and make sure you understand the examples in that chapter.
  - One common question asked when doing SQL interviews is to provide an interview with different datasets and ask how using LEFT, RIGHT, INNER or OUTER command will effect a join.
  - I have never heard of an interview that used a CROSS JOIN.

- **More advanced syntax:** The most complex queries I have seen in interviews ask questions about NULL, HAVING or require the use of a CASE statement. In particular, questions like "how to make this long dataset wide" or, "when we sort, do NULL values come first or last?"

  To prepare for this level of questions:

  - Review the more advanced questions in each homework assignment.
  - Spend time on the practice exams.

Two final points when doing SQL interviews:

First, when doing an interview, make sure that you understand the data before starting to write the query. Ask questions about what columns are unique, which ones have duplicates and what different data types are in each column (if you do not know or they are not obvious). The most common failure, aside from not knowing any SQL, is making an assumption about the data that is incorrect.

Secondly, be careful when writing the query. Write it nicely on the board/keyboard, use space on the board to make the query easy to read. Do not write this:

```
select count(1) as numsales , name from transaction left join salesperson
using( sid ) group by name order by 1 desc limit 5;
```

DRAFT

Write this instead:

```
select
        count(1) as numsales
        , name
from
        transaction
left join
        salesperson
using( sid )
group by name
order by 1 desc
limit 5;
```

2 Example Interview #1

The following questions were given to a student for an interview for a full-time data focused role at Facebook.[1] This was part of the initial screening and was done on a video call with screen sharing set up. The student was required to open a text editor on their screen, which the interviewer could see. The interview then provided the student with information and a set of queries they were required to answer. All of the above information was provided to the person being interviewed.

Note that there were some fairly straightforward warm-up questions (e.g. write a query which returns all rows and columns about a particular user), but those were not recorded below.

The information in the table is about a Facebook product called *Community Translator* which allows people to provide translations or vote on translations. The table is a log of actions that are taken by users and has the following form:

Name	Type	Examples
date	STRING	format - 2019-03-31
user_id	BIGINT	format - 81238123
language	STRING	Arabic, Spanish, Swahili, etc.
device	STRING	2 possible values - desktop OR mobile
action	STRING	2 possible values - vote OR translate

Questions

1. What were the top 10 languages yesterday, in terms of number of unique users who were translators?
2. Return a table with the number of translations completed per language.
3. How many users submitted more than one action yesterday?
4. How many users voted yesterday but didn't translate yesterday?

Answers

1. What were the top 10 languages yesterday, in terms of number of unique users who were translators?

[1] While this *exact* student was interviewing for a data analyst level position, multiple people have coorobated that this is done for data science roles.

DRAFT

```
select
        language
from
        table
where action = 'Translate'
order by count(distinct user_id) desc
limit 10;
```

2. Return a table with the number of translations completed per language.

```
select
        language, count(1) as ct
from
        table
where action  = 'Translate'
group by 1;
```

3. How many users submitted more than one action yesterday?

```
select count(1) as ct
from
(select
        user_id
from table
where date = date(now()) - 1
group by 1
having count(distinct action ) > 1) as innerQ
```

4. How many users voted yesterday but didn't translate yesterday?

```
select
        count(distinct user_id)
from table
        where
                user_id NOT in
                        (select distinct userid from table
                                where date = date(now()) - 1 and action ='Translate')
        and action = 'Vote' and date = date(now()) - 1;
```

Or using a join:

```
select count(1) as ct
from
(select distinct user_ID from table
        where date = date(now()) - 1 and actions = 'Vote') as lhs
left join
(select distinct user_ID from table
        where date = date(now()) - 1 and actions = 'Translate') as rhs
on lhs.user_ID = rhs.user_iD
where rhs.user_ID is null;
```

Or using a CASE statement:

```
select
        count(1) as ct
from
        (select
                user_id
                , max( case when actions = 'Vote' then 1 else 0 end) as m1
                , max( case when actions = 'Translate' then 1 else 0 end) as m2
        from
                table
        where date = date(now()) -1
        group by 1) as innerQ
where m1 = 1 and m2 = 0;
```

3 Example Interview #2

In this whiteboard interview, the interviewee was given the following information on two tables:

- **orders table:** Contains information on orders and had three columns (order_id (int), vendor_id (int) and order_value (float))
- **returns table:** Contains information on returns and had three columns (return_id (int), order_id (int) and return_reason (string))

Both order_id and return_id were integer ids that incremented on their respective tables. Not all orders had returns.

Questions

1. Top 5 vendors (vendor_id) in terms of the largest order.
2. Return rate (number of returns divided by number of orders) for each vendor. Make sure to return zero as the rate if the vendor has no returns.
3. For each vendor return the most common return reason.

Answers

1. Top 5 vendors in terms of the largest order.

```
select
        vendor_id
from
        orders
group by 1
order by max( order_value ) desc
limit 1;
```

2. Return rate (number of returns divided by number of orders) for each vendor. Make sure to return zero as the rate if the vendor has no returns.

DRAFT

```
select
        vendor_id
        , count( return_id )::float / count(orders.order_id) as return_rate
from
        orders
left join
        returns
on orders.order_id = returns.order_id
group by 1;
```

3. For each vendor return the most common return reason.

   (a) Using Analytic Functions

```
select
        vendor_id, return_reason
from
        (select *
                , max( reason_ct) over(partition by vendor_id
                                order by reason_ct desc
                                rows between unbounded preceding
                                        and unbounded following) as mxct
        from
                (select
                        return_reason
                        , vendor_id
                        , count(1) as reason_ct
                from returns group by 1,2) as iq
        ) as iq2
where maxct = reason_ct;
```

   (b) Using CTE

```
with
        total_return_count as
                ( select return_reason, vendor_id, count(1) as reason_ct
                        from returns group by 1,2)
select
        lhs.vendor_id, rhs.return_reason
from
        (select max( reason_ct) as maxct, vendor_id
                from total_return_count group by 2) as lhs
join
        total_return_count as rhs
where lhs.vendor_id = rhs.vendor_id and lhs.maxct = rhs.reason_ct;
```

   (c) No Analytic Functions, no CTEs:

```
select lhs.return_reason, lhs.vendor_id
from
        (select return_reason, vendor_id, count(1) as reason_ct
                from returns group by 1,2) as lhs
join
        (select vendor_id, max(reason_ct) as maxct
                from
        (select return_reason, vendor_id, count(1) as reason_ct
                from returns group by 1,2) as innerRHS
        group by 1) as RHS
on lhs.vendor_id = rhs.vendor_id and lhs.reason_ct = maxct
```

4 Example Interview #3

This set of questions was given during 2019. Note that these require some knowledge of date functions. There are two tables, each with two columns. Columns with the same name can be assumed to match. Note that it is possible for a person to convert without appearing in the visit table.

- **Conversion Table:** Information on users converting (paying) on a website.
  - **User_id:** The ID of the user (integer).
  - **Converstion_ts:** The timestamp (date and time) of when the conversion occurred.
- **Visit Table:** Information about users visiting an advertising web site.
  - **User_id:** The ID of the user (integer).
  - **vist_ts:** The timestamp (date and time) of when the user visited the site.

1. How many visits did each user have to the website?

```
select user_id, count(1) as num_visits
from
visit
group by 1;
```

2. What were the top-5 users in terms of visits?

```
select user_id
from
visit
group by user_id
order by count(1) desc
limit 5;
```

3. Return a list of users who visited the site today, yesterday and the day before yesterday.

DRAFT

```
select distinct v1.user_id
from
(select distinct user_id from visit where date(visit_ts) = date(now()) ) as v1
join
(select distinct user_id from visit where date(visit_ts) = date(now()) - 1) as v2
on v1.user_id = v2.user_id
join
(select distinct user_id from visit where date(visit_ts) = date(now()) - 2) as v3
on v1.user_id = v3.user_id;
```

4. List the three most recent visit timestamps which occurred before the conversion timestamp. If no visit occurred then return one row with a null timestamp. You can also assume that a user_id has, at most, a single conversion.

   There are a few different ways to answer this question, the first is the most general and relies only upon standard SQL syntax. The basic logic is to first exclude all visits which are after the conversion and then join the visit table on itself counting the number of rows after aggregating down on one side. If the number is less than or equal to 3, you know that the number of visits is within 3.

```
select
        lhs.user_id, lhs.conversion_ts, rhs1.visit_ts
from
        conversion as lhs
left join
        visit as rhs1
on lhs.user_id = rhs1.user_id
        and rhs1.visit_ts <= lhs.conversion_ts
left join
        visit as rhs2
on lhs.user_id = rhs2.user_id
        and rhs2.visit_ts <= lhs.conversion_ts
        and rhs1.conversion_ts <= rhs2.conversion_ts
group by 1,2,3
having count(1) <= 3;
```

5 Example Interview #4

This interview was given in 2023 to a student in UChicago's MACSS program. The interview was conducted with the student sharing their screen. Two tables were described in the interview with the column "SID" connecting them.

- **salespeople** contains information on sales people at a company
  - SID
  - name
- **sales** contains information on sales that were made by each sales person. Note that SID is *not* unique as a sales person can have multiple sales.
  - SID
  - sales_amt

1. Tell me the total amount of sales generated by each sales person (name).

```
select name, sum(sales_amt) as total_sold
from
salespeople left join sales using(SID)
group by 1;
```

2. Tell me the names of salespeople who did *not* have any sales.

   Two options. Depending on the size of the table there could be significant performance differences.

```
select lhs.side
from
select sid from salespeople as lhs
left join
(select distinct sid from sales) as rhs
on lhs.sid = rhs.sid
where rhs.sid is null;
```

```
select sid from salespeople where
sid not in (select distinct sid from sales)
```

DRAFT

Chapter 14

Introduction

DRAFT

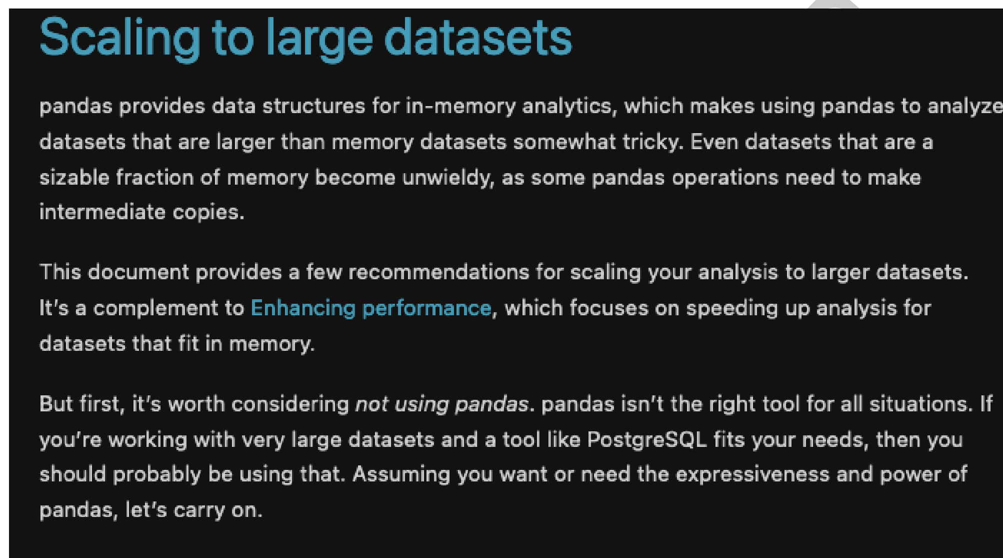
Contents

1 What is Pandas 225
2 Data structures 226
3 Selecting Columns and Rows 231
4 Column Types Conversion 237
5 Dealing with NaN 237
6 Choosing the largest and smallest values 239
7 Manipulating Data & Method Chaining 240
8 Indexes: Creating and Dropping 244
9 Views and Copies 246

DRAFT

1 What is Pandas

- Pandas is a Python library for manipulating data developed by Wes McKinney.
- Pandas itself is a front-end API for manipulating data that is stored *in-memory*. This type of tool is often called an *in-memory database* and, when appropriate, this type of database is the far superior than the relational databases that we have been talking about previously.
- The limitation of this type of database is that the data needs to be small enough to fit into your computer's RAM. For modern data analysis this is a major limitation as many data sets are far larger than the present memory.
- The pandas documentation provides a nice summary of this:[1]



Scaling to large datasets

pandas provides data structures for in-memory analytics, which makes using pandas to analyze datasets that are larger than memory datasets somewhat tricky. Even datasets that are a sizable fraction of memory become unwieldy, as some pandas operations need to make intermediate copies.

This document provides a few recommendations for scaling your analysis to larger datasets. It's a complement to Enhancing performance, which focuses on speeding up analysis for datasets that fit in memory.

But first, it's worth considering *not using pandas*. pandas isn't the right tool for all situations. If you're working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let's carry on.

Figure 14.1: Pandas limitations

- Mentally, pandas consists of a front-end API and a backend in-memory data structure. Before version 2.0 the only backend that was in use was numpy. Specifically complex pandas data structures were built upon numpy based objects.[2]
- The (still default) backend required significant memory use to store data. Per Wes McKinney, *pandas rule of thumb: have 5 to 10 times as much RAM as the size of your dataset*, which is a pretty rough standard to hew to.
- The latest release of pandas, version 2.0, which will be released in *March, 2023* creates a more significant abstraction between the back- and front-end API and allows for the use of Apache Arrow as a backend storage structure in place of numpy.
- Apache Arrow is a columnar in-memory data structure which is optimized for vectorization via SIMD (single instruction, multiple data) routines. By using the Apache Arrow backend, rather than numpy, the ratio to dataset size to memory size is much more manageable.
- The Apache Arrow API is very similar to numpy, but has a number of additional types and also has better support for missing values across types.

[1] Taken from `https://pandas.pydata.org/docs/user_guide/scale.html`

[2] Wes McKinney's blog has a bunch of information about the decision making at the time, you can find it here: `https://wesmckinney.com/blog/apache-arrow-pandas-internals/`

- Pandas is both powerful and useful, especially when doing data analysis. However it comes with some significant downsides:

  1. Pandas is more imperative than SQL which means you tell the interpreter what to do, step-by-step. For data operations there are often many ways to undertake the same operation which means that two people can write two very different Pandas expressions and generate the same results.

  2. Pandas is *not* persistent. It stores data in memory and, if the computer crashes, you lose it.

  3. Pandas is only efficient when the size of data is small enough to "fit" into memory.

  4. Pandas is *not* easily scalable across multiple computers.

- **Since, at the time of the start of this course, version 2.0 has *NOT* been released, we will still be leveraging version 1.5 in the course of this class.**

- To begin we import the Pandas and NumPy packages. By convention, we usually do it as follows:

```
>>> import numpy as np

>>> import pandas as pd
```

- So, when should you use Pandas? You should use it when you have data that easily fits into your computer's memory and you wish to explore or perform straightforward analysis on that data.

2 Data structures

- Starting from the bottom there are a number of data *types*. The chart below shows how they line up with underlying Python and Numpy types.

Description	Pandas Type	Numpy Type	Python (built-in)
Integers	int64	int8, int16, int32, int64, etc.	int
Text	object	string, unicode	str
Double/Float	float64	float16, float32, float64	float
Boolean (T/F)	bool	bool	bool
DateTime	datetime64	datetime64	N/A

Table 14.1: Common Data Types

- The types above are how the underlying data is stored, what operations are allowed on it and how those operations act. For example a "+" will do string concatenation when paired with two objects, but will undertake mathematical addition when paired with floats and integers.

- In Pandas these underlying data types are what we put together to build the two basic data structures that we find in Pandas: DataFrames and Series.

- To determine the type of a particular python object we use the `type` command:

```
>>> type( [1,2,3])
```

  which will return `list` or `<class 'list'>`.

- The type function works on both underlying data types as well as the larger data structures.

DRAFT

- **An important thing to remember:** Pandas is a bit inconsistent in its design and it is frequently the case that a function, operation or something you do will inadvertently change your data. Whenever diagnosing an issue in Pandas, make sure to check the type. You'll often be surprised about what you are working with.
- The way to think about both a Series and a DataFrame is that they are two objects (index and values), combined. The value component contains the actual data while the index component is its own object which has operations that are allowed on it. Note that index objects are complex and can exist in both *rows* and *columns*. We will touch upon this a bit later.

Series

- A Series is one dimensional data object with an associated *index*.
- There are a number of different way to create a list. For example, consider the following two commands which initialize two variables ("x_1" and "x_2") which contain the same values.
- The "name" input defines what the column name is while the list at the start are the values. In the second example, we also have an official "index" which creates labels for the rows of the data.

```
>>> x_1 = pd.Series( [1,2,3,4], name="v1")

>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")
```

- Printing the values yields the following:

```
>>> x_1
0    1
1    2
2    3
3    4
Name: v1, dtype: int64

>>> x_2
a    1
b    2
c    3
d    4
Name: v1, dtype: int64
```

- Note that `x_1` has an index starting from zero and ending at 3 while the second has an index of a, b, c, d.
- We can access the values and index of a series using the following attributes:

```
>>> x_2.values
[1 2 3 4]

>>> type(x_2.values)
<class 'numpy.ndarray'>
```

The returned object is an *array*, not a series! When we ask for the index, we return an "Index" object, which is a type of Pandas data structure that is similar to a data frame.

```
>>> x_1.index
RangeIndex(start=0, stop=4, step=1)

>>> x_2.index
Index(['a', 'b', 'c', 'd'], dtype='object')
```

- To get the size of a Series we can use the “size” or “shape” attributes. Shape is used more frequently as it will return the *shape* in the case of multidimensional objects (size only returns the *number* of objects).

```
>>> x_2.size
4

>>> x_2.shape
(4,)
```

  The “shape” command returns a tuple.

- Not only can the series have a name (such as “v1” in our above examples), but the index can also have a name. This is not something frequently encountered.

```
>>> x_2.name

>>> x_2.index.name = 'alpha'

>>> print(x_2)
alpha
a    1
b    2
c    3
d    4
Name: v1, dtype: int64
```

- Math can be done between Pandas objects:

```
>>> x_1 + x_1
0    2
1    4
2    6
3    8
Name: v1, dtype: int64

>>> x_2 + x_2
alpha
a    2
b    4
c    6
d    8
Name: v1, dtype: int64
```

- Indexes are *incredibly* important because they strongly effect how operators work. In particular, indexes in Pandas align data and when series are added together real effects occur. For example, consider the following:

```
>>> ans = x_1 + x_2
  v1
----
 nan
 nan
 nan
 nan
 nan
[...]
```

If we look at what is returned it is another Pandas Series, with a single column which has the name "v1" and eight values, all of them NaN.[3] Why did this occur? Two reasons:

1. Since the indexes didn't align, the addition operator assumed that the rows did not align and thus the resulting dataset had 8 rows $(4 + 4)$.
2. When adding objects in Pandas, anything added to NaN is equal to NaN.

- If we only want to display the first few data points in a series, we can use the "head" method which can be used as follows:

```
>>> x_2.head(2)
alpha
a    1
b    2
Name: v1, dtype: int64
```

There is also a "tail" method which returns the last rows:

```
>>> x_2.tail(2)
alpha
c    3
d    4
Name: v1, dtype: int64
```

If no integer argument is provided the above methods return 5 values.

- If we want to find out information about what data types are in the series we can use the the `dtypes` variable associated with the object:

```
>>> x_2.dtypes
int64
```

DataFrame

In this section we are going to use the Iowa Cars data (as in the previous SQL section). To use this data, we load the information in a DataFrame, as the command below does:

[3] Interestingly, NaA is from the numpy library, so can be called as "nupy.NaN"

```
>>> dfCars = pd.read_csv('<FILEPATH>/iowa_cars.tdf'
                 ,sep='\t', engine='python', names=['year', 'countyname'
                          , 'motorvehicle' ,'vehiclecat', 'vehicletype'
                          , 'tonnage', 'registrations', 'annualfee'
                          , 'completecategory'])
```

Note that <FILEPATH> needs to be set to your local copy of file.

- A DataFrame is two dimensional data object (think of a table) and an associated index.
- *head, tail, shape, size, index* and *dtypes* are all available and behave as you'd expect:

```
>>> dfCars.head()
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                           [...]
  2011  Jasper        Yes             Moped         Moped                         [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons        [...]
  2015  Palo Alto     No              Trailer       Travel Trailer                [...]
  2016  Adair         Yes             Truck         Truck           3 Tons        [...]

>>> dfCars.shape
(41202, 9)

>>> dfCars.size
370818

>>> dfCars.dtypes
year                  int64
countyname           object
motorvehicle         object
vehiclecat           object
vehicletype          object
tonnage              object
registrations         int64
annualfee           float64
completecategory     object
dtype: object

>>> dfCars.index
RangeIndex(start=0, stop=41202, step=1)
```

- Outside of `dtypes`, which describes the type of data in each column, the method `describe()` creates a set of summary statistics for all columns:

```
>>> dfCars.describe()
               year  registrations     annualfee
count  41202.000000   41202.000000  3.825800e+04
mean    2013.724504    1767.522135  2.122614e+05
std        4.697944    7417.077934  1.181381e+06
min     2005.000000       1.000000  0.000000e+00
25%     2010.000000      25.000000  3.020000e+03
50%     2014.000000     183.000000  2.464500e+04
75%     2018.000000     989.750000  9.136975e+04
max     2021.000000  218975.000000  4.964507e+07
```

- Note that addition with DataFrames and Series can behave unexpectedly. Consider the following

examples:

```
>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")

>>> x_3 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v2")

>>> d_2 = x_2.to_frame()

>>> d_3 = x_3.to_frame()
```

We now have two series (`x_2` and `x_3`) which are the same except for the name ("v1" vs. "v2") and we have two DataFrames based off of those series. Let's do some math:

```
>>> x_2 + x_3
a    2
b    4
c    6
d    8
dtype: int64

>>> d_2 + d_3
   v1  v2
a NaN NaN
b NaN NaN
c NaN NaN
d NaN NaN
```

In the first example we can see that the columns are combined, even though their names are different, but in the second we up with a DataFrame with two NaN columns. In other words, operations on a Series ignore the name while in the DataFrame they do not!

3 Selecting Columns and Rows

- There are a *ton* of ways to select rows and columns using Pandas.
- When I first learned Pandas this was the most frustrating part of the experience. In order to minimize the frustration, I recommend focusing on identifying exactly what data object you have and what data object you want returned and then learning only one method for accessing data in that fashion.
- To specify rows and columns, from a DataFrame, we will use `loc` and `iloc` methods.[4]
- To specify objects, there are a number of different notational devices that can be used:
  - For a single column we can use "dot" notation, with a column name after the dot, which returns a series. Note that this only works if the column name is *not* a reserved word for a DataFrame.

```
>>> type( dfCars.tonnage )
<class 'pandas.core.series.Series'>
```

  - Note that a "dot" can be used with any named object, such as an index:

[4] Another one, "ix", was common but is now deprecated and will not be acceptable in this course.

DRAFT

```
>>> x_2.b
2
```

– We can also specify a column using square brackets:

```
>>> type( dfCars['tonnage'] )
<class 'pandas.core.series.Series'>
```

– We can also use double square brackets:

```
>>> type( dfCars[['tonnage']] )
<class 'pandas.core.frame.DataFrame'>
```

Look at the difference between the previous two examples: the first returned a Series while the second returned a DataFrame. We will find that this convention routinely reappears when using pandas – supplying an itterable as a selector returns a DataFrame while an atomic value returns a Series.

– We can also use `loc`, though to select all rows we need to prepend a colon:

```
>>> type( dfCars.loc[:, 'tonnage'] )
<class 'pandas.core.series.Series'>

>>> type( dfCars.loc[:, ['tonnage']] )
<class 'pandas.core.frame.DataFrame'>
```

– Keep in mind is that using `loc` is the preferred method for accessing data based on contents. One issue that can occur when using alternative methods, such as only using the `[[column]]` is that Pandas can interpret this as a *row* specification, rather than column name. No error is returned when this occurs – instead an empty DataFrame is returned as no rows match that definition.

– In this course **you should always use `loc` when selecting rows and columns without leveraging an index**.

– If there is an index, we can specify rows by using loc with the index value:

```
>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")

>>> x_2['b']
2
```

– **Strong recommendation: specify both rows and columns when selecting.** The reason for this is that when reading something like `df.loc['value']` or `df['value']` it's difficult to know if this is referring to a row with an index or a column with a name.

– We can use `iloc`, which is an *integer* based access for rows and columns, though this requires knowing that tonnage is the 7th column:

DRAFT

```
>>> type( dfCars.iloc[:, 7] )
<class 'pandas.core.series.Series'>

>>> type( dfCars.iloc[:, [7]] )
<class 'pandas.core.frame.DataFrame'>
```

– The `iloc` method takes any standard *slice* when it is used and can refer to both rows and columns. For example, the code below selects the first 10 columns and rows 20 through 50.

```
>>> d_1 = dfCars.iloc[20:50, 0:10]

>>> type(d_1)
<class 'pandas.core.frame.DataFrame'>

>>> d_1.shape
(30, 9)
```

When looking at the above, some of the methods return DataFrames while others return series. **Keep in mind *what* is being returned! Many of the issues with Pandas that user's encounter is due to an unexpected return type!**

- When you start with a Series there is no need to select columns because there is only a single column.
- Now that we know how to convert from a DataFrame to a Series, how do we go the other direction? To do this we, we use the command `to_frame` on the series:

```
>>> type(dfCars.loc[:, 'year'])
<class 'pandas.core.series.Series'>

>>> type(dfCars.loc[:, 'year'].to_frame())
<class 'pandas.core.frame.DataFrame'>
```

DRAFT

Value Counts

- A really useful *Series* method is `value_counts` which returns another series containing the unique values and the number of occurrences of that value within the series:

```
>>> dfCars.loc[:, 'tonnage'].value_counts()
tonnage
<10000 lbs                  3401
6+ Tons Non-Special Usage   2970
6+ Tons Special Usage       2970
3 Tons                      2747
>10000 lbs                  2217
5 Tons                      2089
4 Tons                      1769
0 Tons                       429
2 Tons                         6
Name: count, dtype: int64
```

- By default the `value_counts` method ignores missing values. If we want to include missing values, the parameter `dropna` is set to False:

```
>>> dfCars.loc[:, 'tonnage'].value_counts(dropna=False)
tonnage
None                        22604
<10000 lbs                   3401
6+ Tons Non-Special Usage    2970
6+ Tons Special Usage        2970
3 Tons                       2747
>10000 lbs                   2217
5 Tons                       2089
4 Tons                       1769
0 Tons                        429
2 Tons                          6
Name: count, dtype: int64
```

Content-based selections

- Probably the most common operation done when analyzing data is selecting rows based on a criteria. When doing this with `loc` we provide a boolean object of the same length as the DataFrame. For example, let's consider the following few lines of code:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[ (dfCarsC.loc[:, 'vehicletype'] == 'Semi Trailer') , ['countyname']]

>>> dfCarsC.head()
countyname
------------
Appanoose
Jefferson
Mahaska
Clay
Greene
```

The first line takes our original cars DataFrame and makes a "copy" of it. Why would we do this? Two reasons:

1. Because we may possibly change our original data and, rather than reload it each time, we can make a copy so that we can always restart.[5]
2. Get in the habit of making copies. As we will see later, unexpected issues with pandas can arise because pandas often creates "views" of a DataFrame, rather than make a copy. It is not uncommon for analysis to be incorrect because a user fails to account for views vs. copies.

The second line has three components:

1. `(dfCarsC.loc [:, 'vehicletype'] == 'Semi Trailer')` This creates an array of True and False values based on the boolean condition. In other words, this returns a series of 41,202 items.
2. `dfCarsC.loc[]` The `loc` method, when used in this manner, only returns those rows which the resulting condition evaluate True, so this is going to keep 1,683 rows.

[5]This is good practice when doing exploratory data analysis as it allows you to quickly restart if you do something incorrect.

3. ['countyname'] This portion of the code keeps only the countyname.

Note the following:

- Pandas uses the double equal "==" to do *comparison*. Keep this in mind!
- Why did we have to re-assign the variable (dfCarsC =) in the second line? We had to reassign this because the loc command does not do *in-place* changes. If we didn't do this re-assignment dfCarsC would not have changed.
- This returns a DataFrame and not a Series since we selected the column with a list. If we, instead, used 'countryname' not in a list form we would get a 1 by 1,683 Series instead of a series of 1,683.
- Put all boolean objects in parenthesis when using Pandas!!! Just get used to it, evaluation precedence is AND/OR above equality and inequality, meaning that things will go badly if you don't.
- This example has nested a number of different "things" together. One way to make sure that your Pandas code is readable is by making sure not to overload too many operations into a single line of code.

- Let's do another one! How about we select all columns and only those rows with registrations larger than 10,000? Unsurprisingly, basic row-by-row math operations also work fine.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 10000), :]

>>> dfCarsC
  year  countyname    motorvehicle    vehiclecat    vehicletype            tonnag [...]
------  ------------  --------------  ------------  ---------------------  ------ [...]
  2010  Johnson       Yes             Truck         Truck                  3 Tons [...]
  2008  Cerro Gordo   Yes             Automobile    Automobile                    [...]
  2006  Wapello       Yes             Automobile    Automobile                    [...]
  2009  Black Hawk    No              Trailer       Small Regular Trailer         [...]
  2012  Plymouth      Yes             Automobile    Automobile                    [...]
[...]
```

- We can combine multiple conditions using "and" (&), "or" (|) and not (∼). For example, if we want to get all data from Wright county with more than 1,200 registrations we can run the following commands:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 1200)
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :]

>>> dfCarsC.head()
  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2015  Wright        Yes             Truck          Truck                  3 Ton [...]
  2005  Wright        Yes             Automobile     Automobile                   [...]
  2015  Wright        Yes             Multi-purpose  Multi-purpose                [...]
  2008  Wright        Yes             Truck          Truck                  3 Ton [...]
  2011  Wright        No              Trailer        Small Regular Trailer        [...]
```

Annoyingly these lines are getting longer and longer which makes them more and more difficult to read. To get around this problem, we can add parenthesis around the expression. Adding a

parenthesis allows us to arbitrarily put hard returns and tabs in the expression for organizational purposes:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
                .loc[(dfCarsC.loc[:, 'registrations'] > 1200)
                    & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :])

>>> dfCarsC.head()
  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2015  Wright        Yes             Truck          Truck                  3 Ton [...]
  2005  Wright        Yes             Automobile     Automobile                   [...]
  2015  Wright        Yes             Multi-purpose  Multi-purpose                [...]
  2008  Wright        Yes             Truck          Truck                  3 Ton [...]
  2011  Wright        No              Trailer        Small Regular Trailer        [...]
```

- We can also nest multiple layers of logic using parenthesis:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC.loc[(
        ((dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000))
        |
        ((dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200))
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :])
```

which will return all rows from Wright county with between 1,200 and 3,000 registrations or 4,000 and 4,200 registrations.

- Keep in mind when writing these commands is that the result of what is inside `loc` needs to be a list of true/false boolean expressions *of the same length as the DataFrame.*

- We can combine these operations to filter both on rows and columns:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC.loc[(
        ((dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000))
        |
        ((dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200))
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), 'countyname'])
```

The above will return a Series only containing the `countyname` while the below will be a DataFrame with `countyname` and the number of registrations:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(
        (dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000)
        |
        (dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200)
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), ['countyname','registrations']]
```

DRAFT

4 Column Types Conversion

- We occasionally need to switch types on columns. To do this we use the `astype` method. Consider the following example below:

```
>>> dfCars.loc[:, 'registrations'].dtype
int64

>>> dfCars.loc[:, 'registrations'].astype(str).dtype
object

>>> dfCars.loc[:, 'registrations'].astype(float).dtype
float64
```

- The `astype` is most commonly used when converting between integers, floats and strings. While there is some ability for the `astype` method to be used when dealing with dates and date related objected it is not recommend and there are better, most consistent options available.

5 Dealing with NaN

- Consider the following code:

```
>>> dfCarsC = dfCars.copy()

>>> dfCars217 = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 217000)
    ,['year', 'registrations', 'annualfee']]

>>> print(dfCars217)
       year  registrations   annualfee
2213   2016         217540  32035419.0
2372   2006         218883         NaN
12890  2015         218975  32058351.0
15971  2014         218211  31790136.0
21352  2005         218235         NaN
25896  2008         217073  24160802.0
```

We can see that in this DataFrame we have encountered a number of NaN values which is how missing values are identified in Pandas. To locate these types of values we *do not* use equality operators, but instead the Series method `isna` as shown below:

```
>>> dfCars217.loc[:, 'annualfee'].isna()
2213     False
2372      True
12890    False
15971    False
21352     True
25896    False
Name: annualfee, dtype: bool

>>> dfCars217.loc[:, 'annualfee'] > 1
2213      True
2372     False
12890     True
15971     True
21352    False
25896     True
Name: annualfee, dtype: bool

>>> dfCars217.loc[:, 'annualfee'] < 1
2213     False
2372     False
12890    False
15971    False
21352    False
25896    False
Name: annualfee, dtype: bool
```

- `isna` can also be applied to entire DataFrames:

```
>>> dfCars.isna()
  year    countyname    motorvehicle    vehiclecat    vehicletype    tonnage    r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          0      [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          0      [...]
[...]
```

As with SQL, NaN values evaluate false when given mathematical comparisons. In the example above the NaN values were false for both less-than and greater-than one.

- To identify rows which are not NaN we can use a "not" operator in the following manner:

```
>>> dfCars217.loc[ ~(dfCars217.loc[:, 'annualfee'].isna()), :]
       year  registrations   annualfee
2213   2016         217540  32035419.0
12890  2015         218975  32058351.0
15971  2014         218211  31790136.0
25896  2008         217073  24160802.0
```

- **There is no NaN value for integers**. Any operation on an integer column which generates a NaN

will automatically turn it into a float.

```
>>> x_1
0    1
1    2
2    3
3    4
Name: v1, dtype: int64

>>> x_1.dtypes
int64

>>> x_1[3] = np.nan

>>> print(x_1)
0    1.0
1    2.0
2    3.0
3    NaN
Name: v1, dtype: float64

>>> x_1.dtypes
```

- For NaN values we can use `fillna` in order to replace those values with another:

```
>>> dfCars.loc[:, 'countyname'].fillna('No County')
```

`fillna` can also be used with two columns.

```
>>> dfCars.loc[:, 'tonnage'].fillna(dfCars.loc[:, 'vehiclecat'])
tonnage
---------
Bus
Moped
3 Tons
Trailer
3 Tons
[...]
```

6 Choosing the largest and smallest values

- In the next module we will talk about sorting DataFrames, but for now we are going to cover how to return only certain rows of a DataFrame based on their order. Aay that we want to return the largest annualfee value in Scott county. To do this we use the `nlargest` method which takes two input variables: the number to return and the *series* to sort by.

```
>>> dfCarsC = dfCars.copy()

>>> (dfCarsC.loc[(dfCarsC.loc[:, 'countyname'] == 'Scott'), :]
    .nlargest(1, 'annualfee').loc[:, 'annualfee'])
33497    19235858.0
Name: annualfee, dtype: float64
```

We can do the same thing with smallest values:

```
>>> dfCarsC = dfCars.copy()

>>> (dfCarsC.loc[(dfCarsC.loc[:, 'countyname'] == 'Scott'), :]
    .nsmallest(1, 'annualfee').loc[:, 'annualfee'])
2595    0.0
Name: annualfee, dtype: float64
```

Two important features of the above:

1. The column `'annualfee'` has NaN values in it, but those values were *ignored* in both the smallest and largest operator.
2. We were able to append `.loc[:, 'annualfee']` to the end of the statement in both cases *after* the largest/smallest method. How could we do that? Once again, it has to do with understanding what is being returned. In this case, the method returns a DataFrame which we can then apply any of our selection methods.

7 Manipulating Data & Method Chaining

- So far we have only taken data, filtered rows and selected columns. In this section we are going to do some basic manipulation of the underlying DataFrame.
- Let's start by creating a column, which we can do by using some of the previous selection operators, but this time with a new name:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.loc[:, 'newcol1'] = 5

>>> dfCarsC.loc[:, 'newcol1'].head()
0    5
1    5
2    5
3    5
4    5
Name: newcol1, dtype: int64
```

- Mathematical operations work on a row-by-row basis:

DRAFT

```
>>> dfCarsC.loc[:, 'newcol2'] = dfCarsC.loc[:, 'registrations'] + 5

>>> dfCarsC.loc[:, ['registrations', 'newcol2']].head()
   registrations  newcol2
0              5       10
1            198      203
2           5020     5025
3            366      371
4           2507     2512
```

- Creating columns based on other columns (of the same shape) is also straightforward:

```
>>> dfCarsC.loc[:, 'newcol3'] = dfCarsC.loc[ :, 'registrations'] / dfCarsC.loc[:, 'annualfee']

>>> dfCarsC.loc[:, 'newcol4'] = dfCarsC.loc[:, 'registrations'] * dfCarsC.loc[:, 'annualfee']

>>> dfCarsC.loc[~(dfCarsC.annualfee.isna())
    ,['registrations', 'newcol3', 'newcol4',  'annualfee']].head()
   registrations   newcol3       newcol4  annualfee
0              5  0.007353  3.400000e+03      680.0
1            198  0.142857  2.744280e+05     1386.0
2           5020  0.016201  1.555457e+09   309852.0
3            366  0.019877  6.739158e+06    18413.0
4           2507  0.018752  3.351608e+08   133690.0
```

- Lets take a look at that special case where we divide by zero:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCars.loc[:, 'annualfee'] == 0), ['annualfee', 'registrations']].head()

>>> dfCarsC.loc[:, 'registrations'] / dfCarsC.loc[:, 'annualfee']
2365    inf
2595    inf
2819    inf
2842    inf
3087    inf
dtype: float64
```

We can see that no error was generated *but* a new value `np.inf` was used. Unlike missing values, standard operators can be used on this special value:

```
>>> np.NaN == np.NaN
False

>>> np.inf == np.inf
True

>>> np.inf > 1
True

>>> np.inf < 1
False
```

- The other way to create a new column is by using the `assign` method. Using the `assign` method, rather than the `loc` syntax above allows us to "method chain", because it returns a DataFrame

which has all DataFrame methods available to it:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
    .assign(newcol3=dfCarsC.loc[:, 'registrations']/dfCarsC.loc[:, 'annualfee']))
```

In the example above we use the assign operator to create a new column which is the registrations divided by the annual fee. This is exactly the same result as previously, but this time we applied a more functional method to complete this task.

- The `assign` method also works when setting static values, such as in the following example, which creates a column of zeros. It can also be used with multiple column assignments, as shown below, by separating them with commas.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
    .assign(newcol3=dfCarsC.loc[:, 'registrations']/dfCarsC.loc[:, 'annualfee'], newcol4 = 0)
    .nlargest(5, 'newcol3')
    )
```

- In general we will try to use method chaining methods to manipulate our data because it leads to much more readable code.

Dropping, Renaming and Inplace methods

- We can also drop, or remove, columns from a DataFrame using the `drop` method on a DataFrame. There are two common ways of using this method, both shown below:

```
>>> dfCarsC = (dfCars
                .copy()
                .assign(newcol1 = 1, newcol2=2, newcol3=3, newcol4=4)
                )

>>> dfCarsC = dfCarsC.drop(['newcol1', 'newcol2'], axis=1)

>>> dfCarsC = dfCarsC.drop(columns=['newcol3', 'newcol4'])

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

The first is to provide a list and se the `axis` option to 1 lets Pandas know that we are dropping columns and not indexed rows. The `axis` variable is used to specify if an operation should be done along rows or columns.[6]

The second way of using this method is by specifying the named parameter `columns` and providing a list of strings.

The `drop` method returns a DataFrame with the chosen objects removed and does not modify the

[6] While it seems obvious that we are trying to drop columns in the example, it's just as possible that we have an index and we could be trying to drop rows instead. This variable allows us to be certain which we are doing.

current DataFrame. If we wish to change the current DataFrame (and return `None`) then we use the `inplace` option to the method:

```
>>> dfCarsC = (dfCars
                .copy()
                .assign(newcol1 = 1, newcol2=2, newcol3=3, newcol4=4)
                )

>>> dfCarsC.drop(['newcol1', 'newcol2'], axis=1, inplace=True)

>>> dfCarsC.drop(columns=['newcol3', 'newcol4'], inplace=True)

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

- DataFrame methods commonly have an `inplace` option, which can be helpful to use and easier to read. Be wary however, as it is easy to run into "weird" errors when mixing both `inplace` and not `inplace` commands. In my experience you should either always use `inplace` or never. Mixing them causes problems.
- There are two common ways to rename columns. We can use the `rename` method with the `column` parameter set or we can reassign the labels by setting the `columns` attribute on the DataFrame object.
- The second, setting the `columns` attribute, works because the attribute itself is accessible and while it is shown as an *index* type, it can be set by using any itterable object, such as a list.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

- The columns attribute itself is also an *index* type, but it can be overwritten with any list type to rename the columns. For example:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.columns = ['year2', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory']

>>> dfCarsC.loc[:, 'year2'].head()
  year2
-------
   2008
   2011
   2012
   2015
   2016
```

The column "year" was renamed by changing the name in the list. The object associated with `columns` is *not* mutable and thus must be overwritten and not modified.

DRAFT

- The other way to rename a column is via the `rename` method. `rename` takes a parameter named `column` which we set equal to a dictionary of the form `{old_column : new_column}`

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.rename(columns={'tonnage' : 'tonnage2'})
```

One truly annoying thing about the Pandas rename function is that if you forget the `columns=` parameter, then no error is returned and no column is renamed! In this situation, the command looks for an index value to rename and doesn't find it, so no rename occurs. In other words, the `columns` parameter works the same way that the `axis` parameter did in other commands.

Like many of the other methods, this one also has an "inplace" operator, so the following is equivalent to the previous operation:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.rename(columns={'tonnage' : 'tonnage2'}, inplace=True)
```

- We can use dictionary notation to do multiple renames at the same time using this method:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.rename(columns={'year' : 'year2', 'tonnage' : 'tonnage2'})
```

8 Indexes: Creating and Dropping

- So far our discussion has avoided any mention of how indexes are used in Pandas. In this section we discuss some of the basic use of indexes on the row-level.
- The most common type of index is a `RangeIndex`, which is a simple integer increment. This is the default index set upon loading a dataset:

```
>>> dfCars.index
RangeIndex(start=0, stop=41202, step=1)
```

- To convert a column into an index we use the `set_index` method which takes in a list of columns and creates an index based upon it. For example, the following returns a dataset with an index associated with the "countyname" (which has many repeating values):

```
>>> dfCarsC = dfCars.set_index('countyname')
```

As with many statements in pandas you can have the effect occur in place, rather than returning a new value:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.set_index('countyname', inplace=True)
```

- Index values need not be unique in pandas!

- Indexes can also be multi-level (or hierarchal). We will talk more about this in 4. To add a multi-level index to a DataFrame we supply a list to the `set_index()` method:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.set_index(['countyname', 'year'])

>>> ans = dfCarsC.head()
                     motorvehicle    vehiclecat    vehicletype     tonnage      r [...]
-------------------  --------------  ------------  --------------  ---------  --- [...]
('Ida', 2008)        Yes             Bus           Bus                            [...]
('Jasper', 2011)     Yes             Moped         Moped                          [...]
('Harrison', 2012)   Yes             Truck         Truck           3 Tons         [...]
('Palo Alto', 2015)  No              Trailer       Travel Trailer                 [...]
('Adair', 2016)      Yes             Truck         Truck           3 Tons         [...]
```

- Multi-level index are represented (when printing) as tuples. We also use tuple-like notation to access them, as we will discuss later.
- We can also remove the index (in that we return it to a column value) by using the `reset_index` method:

```
>>> dfCars.reset_index()
```

When using this method, the column name of the former index is equal to the name of the index (e.g. what can be found in `.index.name` on the DataFrame.)

- If we want to reset the index and then not have the index returned as a column, we use the `drop` parameter:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
        .set_index(['countyname', 'year'])
        .reset_index(drop=True)
        )

>>> dfCarsC.dtypes
motorvehicle         object
vehiclecat           object
vehicletype          object
tonnage              object
registrations         int64
annualfee           float64
completecategory     object
dtype: object
```

In the above example, the original two columns (countyname and year) which we turned into the index no longer appear.

- One caveat is that even if you use `reset_index`, this doesn't create a DataFrame without an index. Instead, a standard range index will be created with each row having an index equal to the current row number.
- Finally, we can assign index values by simply assigning them using an object of equal shape.

DRAFT

```
>>> t = dfCars.head()

>>> t.index = np.arange(0, 2*t.shape[0],2)

>>> t.index
Index([0, 2, 4, 6, 8], dtype='int64')
```

9 Views and Copies

- The second most common problem that people have (after not knowing what is being returned) is failing to be explicit about creating copies of the data and instead operate on a view.
- What do I mean by “view” vs. “copy”? A “view”, sometimes called a “shallow copy” is when we create an object based on another object by creating a pointer to a memory space, rather than creating an explicit copy of that data in memory.
- Why would we do this? Namely speed and memory. A “shallow copy”, sometimes called a “zero copy” is much, much faster and places less of a burden on the CPU.
- The downside of this is that it places a burden on the user to understand when a copy or a view is created. Unfortunately in pandas, those rules are not as explicit as they should be and it is common for “weird” behavior to occur because of this.
- We are going to look at five examples of how this behavior can arise.
  1. **Simple View:** In the following example we can see that pandas has created a view of the underlying object.

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df2 = df

>>> df3 = df.iloc[0:1, 0:1]

>>> df.loc[:, 'a'] = 5

>>> print(df2)
   a  b
0  5  0
1  5  1

>>> print(df3)
   a
0  5
```

  2. **Simple Copy:** In the following example we can see that a copy has been created. We use the `.copy()` method in order to explicitly make a copy of the DataFrame.

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df2 = df.copy()

>>> df.loc[:, 'a'] = 5

>>> print(df2)
   a  b
0  0  0
1  1  1
```

3. **Wut? (pt. 1):**

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df2 = df

>>> df3 = df.iloc[0:1, 0:1]

>>> df.loc[:, 'a'] = 5.1

>>> print(df2)
     a  b
0  5.1  0
1  5.1  1

>>> print(df3)
   a
0  0
```

4. **Wut? (pt. 2):**

```python
df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})
print(df, '\n'*2)

column_a = df.loc[:, "a"]

df.iloc[0, 0] = 5
# column_a is a view so it keeps the 5
print(column_a, '\n'*2)

# Lets change column b in the original df
df.loc[:, "b"] = "new entry"
print(df, '\n'*2)

# Now lets change column A in the df
df.iloc[0, 0] = 10
print(df, '\n'*2)

# no longer a view!
print(column_a)
```

```
   a  b
0  0  0
1  1  1


0    5
1    1
Name: a, dtype: int64


   a          b
0  5  new entry
1  1  new entry


    a          b
0  10  new entry
1   1  new entry


0    5
1    1
Name: a, dtype: int64
```

5. **Wut? (pt.3):** In this example you will get a `SettingWithCopyWarning`. This is called “index chaining” and is one of the reasons that we require use `loc` in this course. It is way too easy to end up in situations where this occurs if you use non-`loc` based access methods.

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> df[df.a == 0]['b'] = 100

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df.loc[ (df.loc[:, 'a'] == 0), 'b']= 100

>>> print(df)
   a    b
0  0  100
1  1    1
```

- What have we learned from the above? You have to be careful with views vs. copies. Assuming one or the other can lead to significant problems and unexpected behavior.
- Use `loc` as a way to access data. It's more likely to put you on a path toward a copy.
- The method `_is_view` should return a boolean on if the object is a view or not. I've had varying degrees of luck with it and don't recommend relying on it.
- Use `copy` frequently when you are not intentionally maintaining a view – and maybe most importantly, never assume what you are working on.

DRAFT

DRAFT

Chapter 15

More Manipulations and Types

DRAFT

Contents

1 Sorting DataFrames **253**

2 Dealing with Duplicates **256**

3 Using Type specific functions **258**

3.1 Dates 258

3.2 Strings 260

4 CASE style statements and the "isin" operator **264**

5 Regex Pattern Matching **265**

DRAFT

1 Sorting DataFrames

- If we want to sort a DataFrame or Series then we use the `sort_values` method, which returns the same object with the values sorted.
- This method returns a DataFrame or Series in-which the data is sorted.
- If you are sorting based on a single column you can pass the column name in directly:

```
>>> dfCars.sort_values('annualfee')
```

  If you wish to sort based on multiple columns, you pass them in as a list:

```
>>> dfCars.sort_values(['countyname', 'annualfee'])
```

- The default sort order is ascending (lowest to highest), but we can change that using the "ascending" parameter. This needs to be a boolean (or list of booleans) with a length equal to the number of columns being sorted. Two examples below:

```
>>> dfCars.sort_values(['countyname', 'annualfee'], ascending = [False, True])

>>> dfCars.sort_values('countyname', ascending = False)
```

- The above two queries *do not modify the original DataFrame*, but only return a sorted version. If we wish to modify the original DataFrame we have to either use another assignment operator or use the `inplace` argument:

DRAFT

```
>>> d_1 = dfCars.loc[:,['annualfee', 'registrations']]

>>> d_1.head()
   annualfee  registrations
0      680.0              5
1     1386.0            198
2   309852.0           5020
3    18413.0            366
4   133690.0           2507

>>> d_1.sort_values('annualfee').head()
       annualfee  registrations
37907        0.0              1
34713        0.0              1
4454         0.0              1
38246        0.0              3
30700        0.0              1

>>> d_1.head()
   annualfee  registrations
0      680.0              5
1     1386.0            198
2   309852.0           5020
3    18413.0            366
4   133690.0           2507

>>> d_1.sort_values('annualfee', inplace=True)

>>> d_1.head()
       annualfee  registrations
37907        0.0              1
34713        0.0              1
4454         0.0              1
38246        0.0              3
30700        0.0              1
```

- Before proceeding, take a look at the index numbers on the rows above. The first `d_1.head()` call returns an index of 0 through 4. The next two, however, return an index of the numbers that mapped to the original RangeIndex row location (37907, 34713, ...). This means that the index on the DataFrame returned is no longer in order. If we wish to reorder the index so that it matches the actual row number we will need to change the index, as well will see below.
- Note that the above has important implications for how operations work in pandas. Specifically, operations follow the index – not the row order! This is true *even if we do not explicitly specify an index*.

```
>>> d_1 = pd.DataFrame( [1,2,3], columns=['a'])

>>> d_2 = d_1.sort_values(['a'], ascending=False).copy()

>>> d_1
  a
---
  1
  2
  3

>>> d_2
  a
---
  3
  2
  1

>>> d_1 + d_2
  a
---
  2
  4
  6
```

The important feature of pandas to keep in mind is that all operations are index-based, unless specified otherwise.

- In my experience it is most common to want to sort by the internal values of the data which are not an index. If, however, we wish to sort based on the index values, then we can use the `sort_index` command which will return a DataFrame or Series sorted by the index:

```
>>> d_1 = pd.DataFrame({'A' : [1,2,3], 'B' : [3,2,1]})

>>> d_1 = d_1.sort_values(['B'], ascending = True)

>>> d_1
  A    B
---  ---
  3    1
  2    2
  1    3

>>> d_1.sort_index()
  A    B
---  ---
  1    3
  2    2
  3    1
```

You can see that the DataFrame which was returned by the `sort_index` has now been sorted along

that dimension. This command also has an `inplace` argument.

- How are Nulls handled? Unlike SQL which treats Nulls as the largest value and last alphabetically, pandas treats Nulls as separate condition. Specifically, Nulls are always put last. If you want to change this behavior there is a named argument `na_position` which can be set to either `first` or `last` which determines the position of Nulls in the sort order.

```
>>> dfCars.sort_values(['annualfee'], na_position='first').loc[:, 'annualfee'].head()
11   NaN
14   NaN
22   NaN
23   NaN
44   NaN
Name: annualfee, dtype: float64

>>> dfCars.sort_values(['annualfee'], na_position='last').loc[:, 'annualfee'].head()
37907    0.0
34713    0.0
4454     0.0
38246    0.0
30700    0.0
Name: annualfee, dtype: float64
```

2 Dealing with Duplicates

- A frequent work flow when duplicate values occur within a dataset is removing them. Pandas provides a few useful operations in order to do this: `unique`, `drop_duplicates` and `duplicated`.
- The first of these, `unique` works on a Series and returns the unique values within that Series:

```
>>> dfCars.loc[:, 'vehiclecat'].unique()
['Bus' 'Moped' 'Truck' 'Trailer' 'Multi-purpose' 'Motor Home' 'Automobile'
 'Motorcycle' 'Autocycle']
```

This operation returns an `array`, not a Series, so be careful!

- The second operation, `drop_duplicates`, works on both Series and DataFrames and returns the same object that it is called upon. For example, the following operation does the same thing as the previous code snippet, but this time returns a Series:

```
>>> dfCars.loc[:, 'vehiclecat'].drop_duplicates()
0                Bus
1              Moped
2              Truck
3            Trailer
13     Multi-purpose
22        Motor Home
50        Automobile
81        Motorcycle
411        Autocycle
Name: vehiclecat, dtype: object
```

- This operation can also be done on entire DataFrames:

DRAFT

```
>>> dfCars.loc[:, ['vehiclecat', 'year']].drop_duplicates().head()
  vehiclecat  year
0        Bus  2008
1      Moped  2011
2      Truck  2012
3    Trailer  2015
4      Truck  2016
```

This example returns a DataFrame with only 142 rows, which contains the unique values of the two specified columns. As in SQL, this does *not* create data. If there is combination of data points which is not present within the original data that combination will not appear in the result.

- One common operation we want to handle is removing duplicates from a DataFrame based on a subset of the columns in the DataFrame. We can do this using the `drop_duplicates` method with the `subset`argument set to a list of the columns we want to return as unique.
- Dropping duplicates in this matter raises the issue of which, non-de-duplicated rows do we want to keep? Specifically consider the following example below:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = ( dfCarsC
 .loc[(dfCarsC.loc[:, 'countyname'] == 'Polk')
      & (dfCarsC.loc[:, 'completecategory'] == 'Motor Home - A')
      , :]
 .sort_values('year')
)

>>> # SPOT

>>> dfCarsC.drop_duplicates(subset='countyname')
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage      [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2005  Polk          Yes             Motor Home    Motor Home - A               [...]

>>> dfCarsC = dfCarsC.sort_values('year', ascending=False)

>>> dfCarsC.drop_duplicates(subset='countyname')
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage      [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2021  Polk          Yes             Motor Home    Motor Home - A               [...]
```

- We can see that in the above example we use the `drop_duplicates` method and, in the first example it returns the row from 2005 while the second returns the row from 2021. If we were doing an analysis where we wanted the result of this command than we have cognizant of the current row order since this method, by default, keeps the *first row* that it encounters within that DataFrame.
- Looking at that example, consider the spot where there is a comment in the code above. If someone later came and did something which switched the sort order in the DataFrame then the analysis could be changed without registering an error – subsequent analysis would therefore be based on a different DataFrame and return a different number.
- Because of this, I strongly recommend (nay, require), all `drop_duplicates` in code bases I oversee to have their sort order fully specified with method chaining, such as in the below.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = ( dfCarsC
 .loc[(dfCarsC.loc[:, 'countyname'] == 'Polk')
      & (dfCarsC.loc[:, 'completecategory'] == 'Motor Home - A')
      , :]
 .sort_values('year')
)

>>> (dfCarsC
     .sort_values(['countyname', 'year'], ascending=False)
     .drop_duplicates(subset=['countyname'], keep='first')
     )
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage      [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2021  Polk          Yes             Motor Home    Motor Home - A                [...]
```

- In the above I have also added the keep argument to the command. I do this because, while this is the default behavior, I can't (and don't expect) others to remember what is the default. Putting it in works as a reminder.
- A final useful command is duplicated which returns a boolean series which identifies if a particular row is a duplicate or not. Just like the drop_duplicates method it can take both a subset of columns as well as a keep named argument to guide which rows are deemed a duplicate or not.

3 Using Type specific functions

- As can be expected with any data focused library, Pandas has a large number of useful *type specific functions*. Type specific functions are functions which are only allowed to operate on data types which have a particular type. To reach these methods in Pandas we use a set of *Series* methods as accessor attributes.
- In this section we will consider two sets of type specific functions: those that work on Dates and those that work on Strings.

3.1 Dates

- Date and date related objects are built from the standard Python datatime library.
- There are two commonly used data types:
  1. The datetime64 type which represents a discrete date and time.
  2. The timedelta type which represents an interval.
- There are four common operations we want to do with dates:
  1. **Convert string to datetime:** The two most common ways of doing this:
     (a) **Upon Loading:** When the data is being loaded, there are ways to tell Pandas that a particular column is a date. When loading CSV files, the read_csv method has an argument called parse_dates which accepts a list of columns which are converted upon loading. Note that this method tries to infer the date from the data present and frequently fails.
     (b) **Via to_datetime:** Using this method entails either replacing a column or assigning a new column using this built-in Pandas method:

```
>>> dfMTA.loc[:, 'mtadt'] = pd.to_datetime( dfMTA.loc[:, 'mtadt'], format='%m/%d/%Y')
```

Both the `to_datetime` method as well as a number of other date methods require the user to specify the explicit format of the date string to be processed using the *strftime/strptime* formatting system. This system is a common method of defining the date type in data processing and is based upon an older unix/C standard.[1] The function strftime function takes a date object and returns a string while the strptime takes a string and returns a date object.

The format uses a set of conversion characters, which begin with a % to represent parts of a date. These conversion characters and then combined with ordinary characters (everything else) to combine to specify the final date format. For example, consider the following specification: `%Y-%m-%d` which would represent the year, month and date, separated by dashes. You can see further examples in Table 15.1 below.

Syntax	Example	Description
`%Y-%m-%d`	2022-02-02	Four-digit year and zero padded month and day
`%m/%d/%Y %H:%M:%S`	03/02/2023 14:55:22	M/D/Y (all zero padded) and hour/min/sec with a 24 hr clock

Table 15.1: strftime examples

The Python documentation has a full list of acceptable codes. That documentation can be found here: `https://docs.python.org/3/library/datetime.html#strftime-and-strptime-for`

2. **Extract a date component:** Once you have a datetime object, you can use the `db` accessor to extract a portion of the date from it. In this case we use the `dt` followed by what we are trying to extract:

```
>>> dfMTA.loc[:, 'mtadt'].dt.year.head()
0    2015
1    2015
2    2015
3    2015
4    2015
Name: mtadt, dtype: int32
```

Options for this include things like `year, month, day` and `dayofweek`.

3. **Convert datetime to string:** As discussed above we can use the `strftime` style formatting with the `dt` accessor function. For example:

```
>>> dfMTA.loc[:, 'mtadt'].dt.strftime('%Y-%m').head()
0    2015-11
1    2015-11
2    2015-11
3    2015-11
4    2015-11
Name: mtadt, dtype: object
```

DRAFT

[1] You can find a link to the man page here: `https://man7.org/linux/man-pages/man3/strftime.3.html`

4. **Basic date math and comparisons:** To add and subtract times we use a `Timedelta` object, which represents a length of time to be applied.

```
>>> (dfMTA.loc[:, 'mtadt'] + pd.Timedelta(days=2)).head()
0   2015-11-30
1   2015-11-30
2   2015-11-30
3   2015-11-30
4   2015-11-30
Name: mtadt, dtype: datetime64[ns]
```

`Timedelta` objects allow us to consistently do math on datetime objects, so use these rather than relying on other methods. Comparison operators ($=, >, <$) work as expected:

```
>>> dfMTA.loc[(dfMTA.loc[:, 'mtadt'] > '2015-01-01')].head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash
0      1 2015-11-28   0         I         477           205
1      1 2015-11-28   0         O         486           252
2      1 2015-11-28   1         I         350           171
3      1 2015-11-28   1         O         307           182
4      1 2015-11-28   2         I         280           133
```

- Unlike in SQL there is no obvious `date_trunc` method. While one might expect the `floor` command to do this, there is an open issue on pandas:

  Pandas is broken: `https://github.com/pandas-dev/pandas/issues/15303#`

- The current date and time can be found with the following commands:

```
>>> pd.to_datetime('now')
2023-08-14 20:50:19.553795

>>> pd.to_datetime('today')
2023-08-14 20:50:19.553917
```

These commands return the current date and time at the time that the code is run. This is useful when trying to write code which would analyze a rolling time frame, such as "the last 90 days."

3.2 Strings

- To reference string functions in Pandas, we call the `str` accessor on a column and then reference the string function.
- These methods exclude NaN values and usually match standard string methods in Python and Excel. Table 15.2 contains a list of some useful functions that can be found in Pandas.
- Consider the following example, which uppercases the countyname:

DRAFT

Name	Description
lower / upper	Lower or Upper Cases a string
len	Returns the length of the string
strip	Removes white spaces and new line characters from a string
split	Splits a string into a list from a given pattern
startswith / endswith	Returns a boolean based on if the string follows the pattern
contains	Returns a boolean if the string contains a pattern

Table 15.2: String Functions in Pandas

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.loc[:, 'UC'] = dfCarsC.loc[:, 'countyname'].str.upper()

>>> dfCarsC.loc[:, ['countyname', 'UC']].head()
  countyname         UC
0        Ida        IDA
1     Jasper     JASPER
2   Harrison   HARRISON
3  Palo Alto  PALO ALTO
4      Adair      ADAIR
```

In this example, the countyname is referenced as a series and then the `str` accessor is called. After the `str` accessor is called then the string function `upper` is called. This returns the uppercased value.

- If we want to refer to specific portions of a string (such as the first or last character), we use the `str` accessor and then apply our normal slicing operations afterwards. For example, to get the first two characters of a string we could do the following:

```
>>> dfCars.loc[:, 'countyname'].str[0:2].head()
0    Id
1    Ja
2    Ha
3    Pa
4    Ad
Name: countyname, dtype: object
```

As with the rest of Python, strings start with zero and a slice is inclusive on the left side, but not on the right. The object returned is another Series.

- To concat strings together we use the `cat` method within the `str` accessor or a `+` :

DRAFT

```
>>> dfCars.loc[:, 'countyname'].str.cat(dfCars.loc[:, 'countyname'])
0                    IdaIda
1              JasperJasper
2          HarrisonHarrison
3        Palo AltoPalo Alto
4                AdairAdair
                ...
41197          MarionMarion
41198            WorthWorth
41199    WinnebagoWinnebago
41200      DelawareDelaware
41201      HumboldtHumboldt
Name: countyname, Length: 41202, dtype: object

>>> dfCars.loc[:, 'countyname'] + dfCars.loc[:, 'countyname']
0                    IdaIda
1              JasperJasper
2          HarrisonHarrison
3        Palo AltoPalo Alto
4                AdairAdair
                ...
41197          MarionMarion
41198            WorthWorth
41199    WinnebagoWinnebago
41200      DelawareDelaware
41201      HumboldtHumboldt
Name: countyname, Length: 41202, dtype: object
```

- Using the `split` function provides us an interesting application of the object data type. Let's look at the top of the "vehicletype" column in the DataFrame:

```
>>> dfCars.loc[:, 'vehicletype'].head(10)
0               Bus
1             Moped
2             Truck
3    Travel Trailer
4             Truck
5             Truck
6             Truck
7             Truck
8             Moped
9             Truck
Name: vehicletype, dtype: object
```

As can be seen in the result, there are multiple different types of vehicles, some with one word and some with multiple words. If we use the `split` method on this, we will create a list:

```
>>> dfCars.loc[:, 'vehicletype'].str.split(' ').head(10)
0                [Bus]
1              [Moped]
2              [Truck]
3    [Travel, Trailer]
4              [Truck]
5              [Truck]
6              [Truck]
7              [Truck]
8              [Moped]
9              [Truck]
Name: vehicletype, dtype: object
```

Each of the phrases in the original dataset has been turned into a list – and the lists do not have the same number of items! Surprisingly, we can store this in the DataFrame itself and the series will have type "object", though the contents will be a list!

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.assign(newcol = dfCars.loc[:, 'vehicletype'].str.split(' '))

>>> dfCarsC.loc[:, 'newcol'].dtypes
object

>>> type( dfCarsC.loc[:, 'newcol'].iloc[0] )
<class 'list'>
```

- Since the string functions themselves are accessible from any string series, they can be chained together to generate more complex operations:

```
>>> dfCars.loc[:, 'countyname'].str.upper().str.startswith('A')
0        False
1        False
2        False
3        False
4         True
         ...
41197    False
41198    False
41199    False
41200    False
41201    False
Name: countyname, Length: 41202, dtype: bool
```

- We can also use them to locate information via the `loc` function. In the example below we added a `fillna` command since there are some NaNs in the underlying data and Pandas doesn't allow `loc` indexing with NaN's present!

```
>>> dfCars.loc[(dfCars.loc[:, 'tonnage'].str.contains('Tons')).fillna(False), :]
  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2012  Harrison      Yes             Truck         Truck          3 Tons         [...]
  2016  Adair         Yes             Truck         Truck          3 Tons         [...]
  2016  Van Buren     Yes             Truck         Truck          4 Tons         [...]
  2018  Story         Yes             Truck         Truck          3 Tons         [...]
  2019  Cerro Gordo   Yes             Truck         Truck          5 Tons         [...]
[...]
```

Specifically, if we try the command above without the `fillna` present the result will be:

```
ValueError: cannot index with vector containing NA / NaN values
```

- Finally, the `str` accessory function returns a string, so we can use standard string slicing functions to manipulate strings.

```
>>> dfCars.loc[:, 'countyname'].str[0:3].str.upper().head()
0    IDA
1    JAS
2    HAR
3    PAL
4    ADA
Name: countyname, dtype: object
```

4 CASE style statements and the "isin" operator

- To mimic SQL style CASE statements, the `loc` operator can be used.
- For example, lets say that we wish to create a column ("regsize") which is equal to 'Small', 'Medium' or 'Large', depending on if the number of registrations is less than 200, between 200 and 500 and more than 500. One way to accomplishing this:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] < 200), 'regsize'] = 'Small'

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] >= 200)
     & (dfCarsC.loc[:, 'registrations'] < 500), 'regsize'] = 'Medium'

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] >= 500), 'regsize' ] = 'Large'

>>> dfCarsC.loc[:, ['registrations', 'regsize']].head()
   registrations regsize
0              5   Small
1            198   Small
2           5020   Large
3            366  Medium
4           2507   Large
```

- The `loc` method is a bit overloaded within Pandas in the sense that it can be used in a variety of different ways that can be at times confusing. In this case we are using the method to not only isolate

rows and columns, but also to assign them values.[2]

- The series object has an `isin` method which behaves similarly to the "in" clause in SQL. Provided a list of items to match it will return a True/False value depending on if the value is in it or not. If we want to isolate all the data relating to both Adair and Wright counties then we use it in the following manner:

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].isin(['Adair', 'Wright'])), 'countyname']
        .value_counts())
countyname
Adair     410
Wright    410
Name: count, dtype: int64
```

A caveat about `isin` is that will return False when applied to a NaN.

```
>>> (dfCars
 .loc[(dfCars.loc[:, 'tonnage'].isna()), 'tonnage']
 .isin(['list', 'of', 'values'])
 .value_counts())
tonnage
False    22604
Name: count, dtype: int64
```

5 Regex Pattern Matching

- Pandas has a number of built-in, useful string matching methods, such as `startswith`, `endswith` and `contains`, which we mentioned previously.
- For more complex matches, pandas allows you to use what is called Regular Expression (sometimes called *regex*).
- The most common way of doing this is by using the accessor `str` and then applying a regex enabled method, such as `findall` or `match`, though there are others.
- Let's do a simple example:

```
>>> dfCars.countyname.str.findall('Adair').head()
0         []
1         []
2         []
3         []
4    [Adair]
Name: countyname, dtype: object
```

You can see that the object returned a list for row containing the matching substrings. Lets do something a bit more complex:

[2]We will learn more about the issues that this arrises when we get to Section 9

```
>>> dfCars.countyname.str.upper().str.findall('A').head()
0       [A]
1       [A]
2       [A]
3    [A, A]
4    [A, A]
Name: countyname, dtype: object
```

Once again, we can see that the all matching strings were returned, which in this case would be two of the letter "A".

- Regex is *incredibly* powerful and *incredibly* complex. We can do things like case insensitive matching:

```
>>> dfCars.countyname.str.findall('(?i)a').head()
0       [a]
1       [a]
2       [a]
3    [a, A]
4    [A, a]
Name: countyname, dtype: object
```

- The following will identify everything that starts with an upper case "A":

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].str.match('^A.*')), 'countyname']
        .value_counts()
        )
countyname
Adair        410
Appanoose    407
Allamakee    403
Audubon      393
Adams        382
Name: count, dtype: int64
```

- An even more complex example is to get everything that starts with "A" and ends with "s", which will find "Adams", but not the rest of the counties above.

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].str.match('^A.*s$')), 'countyname']
        .value_counts()
        )
countyname
Adams    382
Name: count, dtype: int64
```

- It makes sense to spend some time (not a lot) reading over some regex and you should use any opportunity on the homework to play around with it. I'm never going to ask you anything more than the simple, built-in, stuff on an exam.

- While regex is incredibly powerful for pattern matching purposes, its Achilles heel is that it isn't a

strong standard. It is actually a collection of different standards that have significant overlap. The implication is that it is entirely possible that code that works with "Regex" in one place will not work in another.

DRAFT

DRAFT

Chapter 16

Aggregations

DRAFT

Contents

1 Introduction to the MTA dataset 271
2 Simple Aggregations 271
3 GroupBy Objects 274
4 Advanced Index / Multiindex 279
5 If not indexes... 285
6 Indexing with aggregations, a big Gotcha 286

DRAFT

1 Introduction to the MTA dataset

- In this section we will using the NY MTA dataset, as we did in the section XX. In order to load this dataset, use the following command:

```
>>> dfMTA = pd.read_csv('<FILEPATH>/MTA_Hourly.tdf'
                    , sep='\t', engine='python', names=['plaza', 'mtadt'
        , 'hr', 'direction', 'vehiclesez', 'vehiclescash'])

>>> dfMTA.mtadt = pd.to_datetime(dfMTA.mtadt)
```

where <FILEPATH> needs to be changed to the appropriate local location.

2 Simple Aggregations

- Pandas provides a number of ways to do simple aggregations – which we define as those over the entire dataset. The table below shows the available aggregation functions:

Name	Description
count	Number of non-NaN values
sum	Sum of non-NaN values
mean	Average of non-NaN values
size	Number of rows
median	Median of non-NaN values
quantitle(X)	X-th quantitle of non-NaN values
std	Standard Deviation
var	Variance
min	Min of non-NaN values
max	Max of non-NaN values
prod	Product of non-NaN values
first	First non-NaN value
last	Last non-NaN value
nunique	Number of unique values

- For the purposes of this course, we will only focus on those that match the SQL ones: `min`, `max`, `mean`, `sum`, `count` and `nunique` (essentially count distinct).[1]
- There are five aggregation methods, which we will term either "Simplifying" or "Equal" depending on what gets returned relative to the original object they are applied to:
  1. Using an aggregation function, such as `sum()` (Simplifying).
  2. Using the `agg` command, with a string (Simplifying).
  3. Using the `agg` command, with a list of operators (Equal).
  4. Using the `agg` command, with a dictionary of single strings
     - On a DataFrame (Simplifying).
     - On a Series (Equal).

[1] For a complete list, check out the list of "Computations / Descriptive Stats" in the pandas documentation, which can be found here: `http://pandas.pydata.org/pandas-docs/stable/reference/frame.html`

DRAFT

5. Using the `agg` command, with a dictionary of lists. **Cannot be done on Series.** (Equal).

All of the methods above, save the last one, can be applied to both a DataFrame or a Series. The most important thing to remember is that the object you get out from the operation is dependent on both the input type and the operation itself.

- The operations above which say *simplifying* return a "lower" complexity object then the original one while those that say *equal* return an object of "equal" complexity. In terms of complexity order DataFrames are more complex than Series which are more complex than atomic numbers. So, if you use the second method above on a Series we would expect it to return a number. While if we use the third method on a Series we would expect it to return a Series.

- We will first run through each method demonstrating the complexity change. After that we will talk about the internals of the resulting object.

  1. First method: directly applying an aggregation function:

     – If we start with a **Series** and use this *simplifying* method, we will get a **number**:

     ```
     >>> dfMTA.loc[:, 'vehiclescash'].sum()
     330901032
     ```

     – If we start with a **DataFrame** and use the same, *simplifying* method, we will get a **Series**:

     ```
     >>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].sum()
     vehiclescash      330901032
     vehiclesez       1484674162
     dtype: int64
     ```

  2. Second method: Applying an aggregate function using the `agg` function, but only a single item via a **string**. This is again a *simplifying* method.

     ```
     >>> type( dfMTA.loc[:, 'vehiclesez'].agg('sum') )
     <class 'numpy.int64'>

     >>> type( dfMTA.loc[:, ['vehiclesez']].agg('sum') )
     <class 'pandas.core.series.Series'>
     ```

  3. Third method: Applying aggregate function(s) using the `agg` function, but via a **list**. This is an *equal* method:

     ```
     >>> type( dfMTA.loc[:, 'vehiclesez'].agg(['sum']) )
     <class 'pandas.core.series.Series'>

     >>> type( dfMTA.loc[:, ['vehiclesez']].agg(['sum']) )
     <class 'pandas.core.frame.DataFrame'>
     ```

  4. Fourth method: Applying aggregate function(s) using the `agg` function, but via a **dictionary** where every value in the dictionary is a string.

     – In the case of a Series this is an *equal* method:

DRAFT

```
>>> type(dfMTA.loc[:, 'vehiclesez'].agg({'vehiclesez' : 'sum'}))
<class 'pandas.core.series.Series'>
```

- In the case of DataFrame it is a *simplifying* method:

```
>>> type(dfMTA.agg({'vehiclesez' : 'sum'}))
<class 'pandas.core.series.Series'>
```

5. Fifth method: Applying aggregate function(s) using the `agg` function, but via a **dictionary** where every value in the dictionary is a list. This method only exists on DataFrames, it will **not** work on a Series.

```
>>> type(dfMTA.agg({'vehiclesez' : ['sum']}))
<class 'pandas.core.frame.DataFrame'>
```

- Similar to applying the aggregation functions directly there are different objects that can be returned depending on the form of the input. The final form (columns and indexes) of the returned value is also dependent on the data type and the operation that is completed. When looking at our examples, there are a finite number of possibilities:

1. **Returns a number:** This will return an atomic number.
2. **Return a Series:** Returns a Series, two possible forms:
   (a) Index based on aggregate function name

```
>>> dfMTA.loc[:, 'vehiclesez'].agg(['count','sum'])
count       1165728
sum      1484674162
Name: vehiclesez, dtype: int64
```

   (b) Index based on column name

```
>>> dfMTA.loc[:, ['vehiclesez', 'vehiclescash']].agg('sum')
vehiclesez      1484674162
vehiclescash     330901032
dtype: int64
```

3. **Return a DataFrame:** Returns a DataFrame:
   (a) Index based on aggregate function names (therefore columns are column names)

```
>>> dfMTA.agg({'vehiclesez' : ['count', 'sum'], 'vehiclescash' : 'sum'})
       vehiclesez  vehiclescash
count     1165728           NaN
sum    1484674162   330901032.0
```

- Importantly the table in 17.1 contains a list of how our five operators produce output. In this table the **bolded options** present what is recommended for both coverage and ease of remembering.
- A very useful aggregation is `nunique` which counts the number of unique values in a list:

DRAFT

```
>>> dfMTA.loc[:, ['plaza','hr']].agg(['nunique'])
         plaza  hr
nunique     10  24
```

- Lets answer a quick question about the MTA dataset: What percentage of cars which pass through a toll place use an EZ pass?

```
>>> dfMTA.loc[:, 'vehiclesez'].sum()
    / (dfMTA.loc[:, 'vehiclesez'].sum() + dfMTA.loc[:, 'vehiclescash'].sum())
0.8177431410753236
```

- The command above is relatively straightforward. Each of the three aggregation operations returns an `numpy.int64` object. Traditional addition and division are then applied to get the final answer.

3 GroupBy Objects

- More complex aggregation operations require using the `groupby` method in pandas.
- The `groupby` method is a piece of the "split-apply-combine" pattern for handling subsetted data aggregation. This pattern involves taking a data set and *spliting* it along a dimension (usually values within a column or set of columns) *applying* an operation (such as sum) to similar values within those groups and then *combining* the results. Figure 16.1 presents this visually.

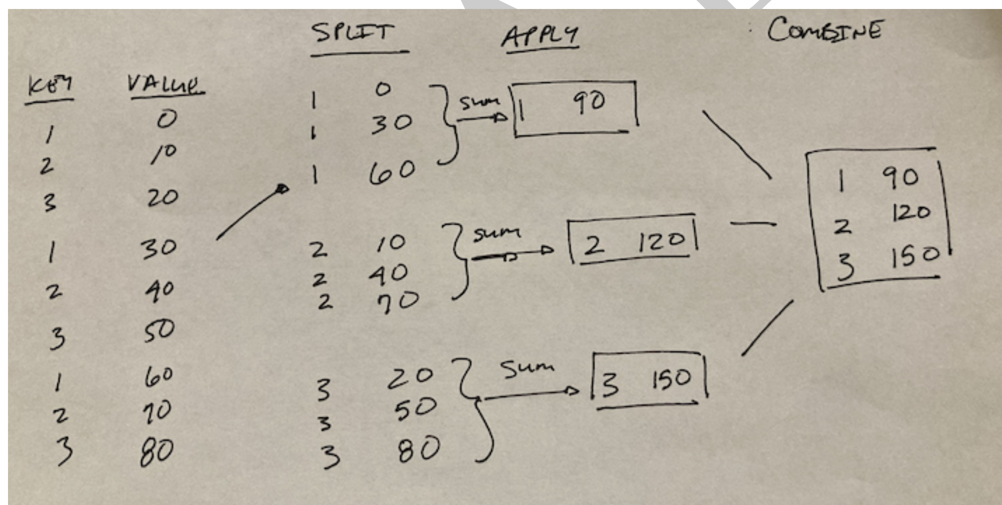


Figure 16.1: Split-Apply-Combine pattern

- For new users of pandas, the `groupby` object can be difficult to understand because it is not a static data result. Instead, the `groupby` object only contains information about the split definition – not the data itself.
- As a first example lets calculate the max `vehiclescash` by `plaza` in the dataset:

```
>>> dfMTAg = dfMTA.loc[:, ['plaza', 'vehiclescash']].groupby('plaza')

>>> type(dfMTAg)
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>

>>> x_1 = dfMTAg.max()

>>> x_1
       vehiclescash
plaza
1              1352
2              1040
3              1594
4              1368
5               674
6               844
7               727
8               599
9              1320
11             2116

>>> type(x_1)
<class 'pandas.core.frame.DataFrame'>
```

The first thing that we do is define a new object `dfMTAg` which is a `groupby` object. Similar to `loc` we provide it with a similar sized object in order to define the grouping. In this case we have told `groupby` to group by `plaza`.

Secondly, we limit our `groupby` object to only the `vehiclescash` column and then use the aggregation function `max` to calculate the maximum value. Instead of the above, we also could have done it this way which doesn't limit the columns calculated:

```
>>> dfMTAg = dfMTA.groupby('plaza')

>>> dfMTAg.max().loc[:, 'vehiclescash']
plaza
1     1352
2     1040
3     1594
4     1368
5      674
6      844
7      727
8      599
9     1320
11    2116
Name: vehiclescash, dtype: int64
```

However, this way would calculate the max across *all* columns before returning the `vehiclescash` column, which is much less efficient.

Note that this has a row index! The result of this calculation is a DataFrame which has an index equal to the columns chosen via the `groupby` method. This would be an index composed of integers.

```
>>> dfMTA.loc[:, ['plaza', 'vehiclescash']].groupby('plaza').max().index
Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 11], dtype='int64', name='plaza')
```

- We can combine multiple operations, such as restricting ourselves to only the second plaza. We either have to do the filtering *before* we create the GroupBy object or do it *after* we have calculated our aggregate functions. Both of the commands below return the same thing, but demonstrate this difference:

```
>>> (dfMTA
     .loc[(dfMTA.loc[:, 'plaza'] == 2)]
     .groupby('plaza')
     .max()
     .loc[:, 'vehiclescash'])
plaza
2    1040
Name: vehiclescash, dtype: int64
```

Once again – *what is being returned in each?* In the first situation we are getting returned a Series with an index (plaza), but only a single row while the second is returning only a single number, in this case a numpy integer.

- How about calculating the percentage of cars using EZ pass over each year?

```
>>> grp = (dfMTA
    .loc[:, ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
    .assign(yr=dfMTA.loc[:, 'mtadt'].dt.year)
    .groupby('yr')
    .sum())

>>> grp.loc[:, 'vehiclesez'] / (grp.loc[:, 'vehiclesez'] + grp.loc[:, 'vehiclescash'])
yr
2010    0.757150
2011    0.792285
2012    0.808791
2013    0.828819
2014    0.836587
2015    0.845386
2016    0.851932
2017    0.851355
dtype: float64
```

This works because the `groupby` object, after the sum has been applied, is just a *standard DataFrame* which we can do whatever expected math we'd like.

- If we want to group by multiple columns at the same time, we put the columns, as a list, into the `groupby` object. For example if we want to know the number of cars using the EZ pass which go through the toll plaza in each direction, we can do the following:

```
>>> d_1 = dfMTA.groupby(['plaza', 'direction']).sum(numeric_only=True)

>>> d_1.head()
                     hr  vehiclesez  vehiclescash
plaza direction
1     I          707112    71598396      26925764
      O          707112    75466906      27433718
2     I          707112   104041531      20567017
      O          707112    81520997      17442388
3     I          707112   104486985      32400024
```

Note the use of the `numeric_only` argument which is set to True. In versions of pandas before 2.0 this was not required, but now it is and if you do not use it a warning will appear. In general pandas will attempt the operation on all allowable columns.

Look carefully at the row index and we can see that we now have a multiindex / hierarchal index on the rows! We will be deep diving into this shortly.

- If we do not want the rows returned as an index we can add the argument `as_index` and set it to False in the `groupby`. Doing this returns the grouping variables as a column, rather than index. It is roughly equivalent to adding a `reset_index` to the result of the aggregation:

```
>>> d_1 = dfMTA.groupby(['plaza', 'direction'], as_index=False).sum(numeric_only=True)

>>> d_1.head()
   plaza direction      hr  vehiclesez  vehiclescash
0      1         I  707112    71598396      26925764
1      1         O  707112    75466906      27433718
2      2         I  707112   104041531      20567017
3      2         O  707112    81520997      17442388
4      3         I  707112   104486985      32400024
```

- Just as with DataFrames and Series there are five methods for doing aggregation and they can either be "simplifying" or "equal" in terms of complexity. In all cases, however, a DataFrame is returned and the simple versus equal refers to the shape of the output data and specifically what is returned in the columns.

  1. Using an aggregation command directly (simplifying).
  2. Using an aggregation command with a string (simplifying)
  3. Using an aggregation command with a list of aggregation functions (equal)
  4. Using an aggregation command with a dictionary of single strings (simplifying)
  5. Using an aggregation command with a dictionary of lists (equal).

- In all cases the row will be an index based on the contents of the `groupby`, so if there is a single item then there will be a single index, if there are multiple items it is a hierarchal/multiindex.
- With `groupby` the distinction between the simplifying and equal operations is how the columns are named / handled and there are two possibilities:

  1. (Similar to *simplifying* methods) We get a DataFrame where the index is the variables in the `groupby` and the columns have the name of the original columns.

DRAFT

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg('sum')

>>> d_1
            hr  vehiclesez
plaza
1      1414224   147065302
2      1414224   185562528
3      1414224   217926485
4      1387176   135142255
5      1414224    46534289
6      1414224    44735980
7      1414224   172800186
8      1412016   106104332
9      1414224   232681943
11      707112   196120862

>>> d_1.columns
Index(['hr', 'vehiclesez'], dtype='object')
```

This method does *not* state what aggregation function was used to get the result.

2. (Similar to *equal* methods) In the more complex case, then the result has a multiindex on the column:

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1
            hr     vehiclesez
           sum max        sum   max
plaza
1      1414224  23  147065302  3066
2      1414224  23  185562528  4307
3      1414224  23  217926485  4572
4      1387176  23  135142255  3640
5      1414224  23   46534289  1747
6      1414224  23   44735980  1604
7      1414224  23  172800186  4042
8      1412016  23  106104332  3402
9      1414224  23  232681943  4926
11      707112  23  196120862  8345

>>> d_1.columns
MultiIndex([(        'hr', 'sum'),
            (        'hr', 'max'),
            ('vehiclesez', 'sum'),
            ('vehiclesez', 'max')],
           )
```

In this case, the object being returned is another DataFrame. A few important notes:

- First, as we saw before the plaza variable has been turned into an index on the rows, as this is the grouping column.
- The columns though are a *total* mess. In this case we have what is called a hierarchal or multi-index on the columns.
- The outer level has the name of the column begin aggregated while the inner level has the aggregation function.
- We can see this index more clearly by looking at the `columns` attribute of the DataFrame which puts a list of tuples as the column information.

- I recommend, when using `gropuby` to lean into the the list/dictionary method as, while the multiindex columns aren't straightforward, it returns the name of the aggregation function that was used:

```
>>> d_1 = (dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
             .groupby( ['hr', 'plaza'] )
             .agg({'vehiclesez' : ['sum', 'max'], 'vehiclescash' : 'max'})
            )

>>> d_1.head()
         vehiclesez       vehiclescash
                sum   max          max
hr plaza
0  1        2535610  1860          900
   2        2626113  1948          556
   3        3521056  2753         1227
   4        1493721  1415          349
   5         480565   430          132
```

4 Advanced Index / Multiindex

- As before we have a DataFrame which has a multi-index on the rows. If we wish to remove this multi-index and return it to a column we can use `reset_index` or use the `as_index` argument in the `groupby` as described previously.

DRAFT

```
>>> d_1 = (dfMTA
    .groupby(['plaza', 'direction'])
    .agg({'mtadt' : ['first'], 'vehiclescash' : ['sum']})
    )

>>> d_1.reset_index(inplace=True)

>>> d_1.head()
  plaza direction      mtadt vehiclescash
                       first          sum
0     1         I 2015-11-28     26925764
1     1         O 2015-11-28     27433718
2     2         I 2015-11-28     20567017
3     2         O 2015-11-28     17442388
4     3         I 2015-11-28     32400024
```

- Before beginning this discussion, I want to preface this by stating that I'm not a big fan of index based methods in pandas and I think that they have some serious limitations.
- In this section we are going to cover how to reference values (both columns and rows) which have mulitindexes.
- The `loc` command can be used to access any index-based method on a **row**. For example:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[2, :]
    )
vehiclesez    sum    185562528
vehiclescash  sum     38009405
Name: 2, dtype: int64
```

In this example the "2" refers to plaza values which are equal to "2" and it gets returned as a Series.

- If we want to return it as a row we can put the selector inside a list:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[[2], :]
    )
      vehiclesez vehiclescash
             sum          sum
plaza
2      185562528     38009405
```

- In some ways this mirrors how we reference columns within `loc` commands. We can use any slice based reference, such as the following two examples demonstrate:

```
>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[2:4, :]
    )
      vehiclesez vehiclescash
             sum          sum
plaza
2      185562528     38009405
3      217926485     67000523
4      135142255     21397862
```

```
>>> (dfCars
    .groupby(['countyname'])
    .agg({'annualfee' : ['sum'], 'registrations' : ['count']})
    .loc[ "A":"B", :])
             annualfee registrations
                   sum         count
countyname
Adair       25300774.0           410
Adams       12645641.0           382
Allamakee   37068964.0           403
Appanoose   29947777.0           407
Audubon     20501452.0           393
```

- Note that in the above we *cannot* put the slice inside a list – this will raise an error:

```
>>> (dfCars
    .groupby(['countyname'])
    .agg({'annualfee' : ['sum'], 'registrations' : ['count']})
    .loc[ ["A":"B"], :])

  File "<stdin>", line 4
    .loc[ ["A":"B"], :])
              ^
SyntaxError: invalid syntax

>>> (dfMTA
    .groupby('plaza')
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[[2:4], :]
    )


  File "<stdin>", line 4
    .loc[ [2:4], :])
              ^
SyntaxError: invalid syntax
```

- As a reminder Python is case sensitive in its sorting:

```
>>> "C" < "D" < "a"
True
```

- In the case of a multi-index we need to use tuples to access rows. In this first example a Series is returned.

```
>>> (dfMTA
    .groupby(['plaza', 'hr'])
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[ (2,4), :])
vehiclesez    sum    1527122
vehiclescash  sum     533945
Name: (2, 4), dtype: int64
```

However, in this case, a DataFrame is returned:

```
>>> (dfMTA
    .groupby(['plaza', 'hr'])
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum']})
    .loc[ [(2,4)], :])
         vehiclesez vehiclescash
                sum          sum
plaza hr
2     4     1527122       533945
```

- Personally, I tend to avoid using these systems as I find them to be complex and filled with a ton of "gotchas", but we will cover it for completeness so that you have seen it.

- Let's consider a specific example using the MTA data as we discuss accessing columns and specifying multiple levels of information.

```
>>> d_1 = (dfMTA
      .groupby(['plaza', 'direction'])
      .agg({'vehiclesez' : ['max', 'min'], 'vehiclescash' : ['max', 'min', 'sum']})
     )

>>> d_1.head()
                vehiclesez     vehiclescash
                       max min          max min       sum
plaza direction
1     I               2962   0         1232   0  26925764
      O               3066   0         1352   0  27433718
2     I               4307   0         1040   0  20567017
      O               3255   0          927   0  17442388
3     I               4572   0         1575   0  32400024
```

- The DataFrame above has an index (plaza and direction) and five columns associated with the aggregations. There are 19 total rows in the DataFrame (the 11th plaza only has Inbound activity). Both the rows and columns have multiindexes.

- To reference objects we can use tuples with our `loc`:

```
>>> d_1.loc[ :, ('vehiclescash', 'max')].head()
plaza  direction
1      I            1232
       O            1352
2      I            1040
       O             927
3      I            1575
Name: (vehiclescash, max), dtype: int64

>>> d_1.loc[ :, [('vehiclescash', 'max')]].head()
                vehiclescash
                         max
plaza direction
1     I                 1232
      O                 1352
2     I                 1040
      O                  927
3     I                 1575

>>> d_1.loc[ (1,'I'), :]
vehiclesez    max        2962
              min           0
vehiclescash  max        1232
              min           0
              sum    26925764
Name: (1, I), dtype: int64

>>> type( d_1.loc[ [(1,'I')], :] )
<class 'pandas.core.frame.DataFrame'>

>>> d_1.loc[ (1,'I'), ('vehiclescash', 'max')]
1232
```

Taking a look at the three examples above, we see that by replacing our traditional column name with a tuple we can reference single objects within our DataFrame.

- The first example above is straightforward and behaves as expected. We pass in a colon to return all rows and then pass in a tuple to identify the columns of interest. This returns a SERIES.
- The second example passes the tuple in as a list and (surprise, surprise), this returns the same data in the previous example, but this time as a DataFrame.
- The third example is similar, expect we use this select rows, rather than columns. HOWEVER, in this case, we find that the object returned is *not* a DataFrame, but instead a Series! This is just because we are selecting a *single* row, which we have completely specified from the original DataFrame. Note that this can happen when selecting rows based off of an index even without using tuples, as seen below.[2]

```
>>> type( dfMTA.iloc[0] )
<class 'pandas.core.series.Series'>
```

[2] Let's call this gotcha #1

- The fourth example applies the same list logic as with columns. In this case, instead of returning a Series, it returns a DataFrame containing the same data as the previous series.
- The fifth example has us selecting both a row and column based on tuples and it returns single value.
- To reference multiple values inside the tuple, we use the command `slice(None)` to create a slice which contains nothing. For example:

```
>>> d_1.loc[ : , ('vehiclesez', slice(None))].head()
                vehiclesez
                       max min
plaza direction
1     I               2962   0
      O               3066   0
2     I               4307   0
      O               3255   0
3     I               4572   0
```

  Or:

```
>>> d_1.loc[ (slice(None), 'I'), (slice(None), 'max')].head()
                vehiclesez vehiclescash
                       max          max
plaza direction
1     I               2962         1232
2     I               4307         1040
3     I               4572         1575
4     I               3640         1368
5     I               1675          674
```

  In this second example we use the tuples to return all the max values in the inbound direction.
- What if we want to select a few different values in a tuple, rather than a single one? We can use a list, which get interpreted as a filter.

```
>>> d_1.loc[ ([1,2], 'I'), (slice(None), 'max')]
                vehiclesez vehiclescash
                       max          max
plaza direction
1     I               2962         1232
2     I               4307         1040
```

  The command above uses a list to select either plaza #1 or plaza #2.
- Note that this tuple logic is *only* when using multiindexes. If you are accessing data based on the contents of the data (and not the index), then the traditional logic we used with `loc` is what you want to use.
- I find the tuple / `slice(None)` logic to be pretty discordant with how I use pandas. Because of this my normal pattern is to avoid using indexes unless there is an operation that requires them. In that case I then `set_index` do the operation and then `reset_index` in order to remove the index. It's too confusing for my small mind.

DRAFT

5 If not indexes...

- I commonly choose to remove the indexes on columns and usually do it via one of the methods below:
  1. **Drop a "level":** This method removes one of the levels (usually the outer one). It's easy to do, but the downside is that the resulting columns may have repeating names. The command to do this is the `droplevel` method of the multi-index and then reassign those values back to the columns:

     ```
     >>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
         .groupby( 'plaza' ).agg(['sum', 'max'])

     >>> d_1.columns = d_1.columns.droplevel()

     >>> d_1.head()
                sum  max        sum   max
     plaza
     1      1414224   23  147065302  3066
     2      1414224   23  185562528  4307
     3      1414224   23  217926485  4572
     4      1387176   23  135142255  3640
     5      1414224   23   46534289  1747
     ```

     The `droplevel` method, without any parameters, drops the outermost level and, in this case, the resulting set of columns have repeating names. If we wish to change those repeated names the best way to do this is by reassigning them via the `columns` parameter.[3]

     ```
     >>> d_1.columns = ['sum_hr', 'max_hr', 'sum_vec', 'max_vec']
     ```

  2. **Concat 'em:** The following piece of code is something that I use frequently in order to remove the multi columns and replace them with a concatenated string.

DRAFT

[3]Since the columns have repeating names, the rename method does not work.

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1.columns = ['_'.join(col).strip() for col
    in d_1.columns.values]

>>> d_1
        hr_sum  hr_max  vehiclesez_sum  vehiclesez_max
plaza
1      1414224      23       147065302            3066
2      1414224      23       185562528            4307
3      1414224      23       217926485            4572
4      1387176      23       135142255            3640
5      1414224      23        46534289            1747
6      1414224      23        44735980            1604
7      1414224      23       172800186            4042
8      1412016      23       106104332            3402
9      1414224      23       232681943            4926
11      707112      23       196120862            8345
```

3. **Blow it all up:** This is the most common solution I use when doing EDA. In this solution I simply keep the columns I'm interested in and rename the rest by setting the column attribute, similar to what we did after the droplevel command earlier:

```
>>> d_1 = dfMTA.loc[:, ['plaza', 'hr', 'vehiclesez']]
    .groupby( 'plaza' ).agg(['sum', 'max'])

>>> d_1.columns = ['sum_hr', 'max_hr', 'sum_vec', 'max_vec']
```

6 Indexing with aggregations, a big Gotcha

- When using `groupby` objects it's important to pay attention to what you are grouping on and, specifically, it is an index or not.
- Frequently we wish the output of a `groupby` to not create an index, but instead just return a standard column. To do this, we use the `as_index=False` option.
- This function is pretty handy because it "seems" to allows us to move data back and forth between index to column when we do a `groupby`.
- HOWEVER, there is a big, big, gotcha with this! That gotcha is that even if you set `as_index=False` it will *not* pull data from an index to a column.
- Consider the following example:

DRAFT

```
>>> d_1 = (dfMTA
    .loc[:, ['mtadt', 'vehiclesez']]
    .groupby('mtadt', as_index=False)
    .sum())

>>> d_1.head()
       mtadt  vehiclesez
0 2010-01-01      316187
1 2010-01-02      380746
2 2010-01-03      359420
3 2010-01-04      494168
4 2010-01-05      518537

>>> d_2 = (dfMTA
    .loc[:, ['mtadt', 'vehiclesez']]
    .set_index(['mtadt'])
    .groupby('mtadt', as_index=False)
    .sum())

>>> d_2.head()
   vehiclesez
0      316187
1      380746
2      359420
3      494168
4      518537
```

In *both* cases we have set `as_index=False`, but in the second case, when the column being grouped is a part of the index *we lose the mtadt*! As stated, this is because when the column is originally an index, the `as_index` method will not pull the column out of the index, instead it will ignore it.

- This can also happen when using `as_index=False` and then using the column in the aggregation. In the example below we have added `as_index=False` and then aggregated by group by column. By doing this, plaza only appears once which, in this case is for the aggregation function and not from a column generated from an index.

```
>>> (dfMTA
    .groupby('plaza', as_index=False)
    .agg({'vehiclesez' : ['sum'], 'vehiclescash' : ['sum'], 'plaza' : 'count'})
    )
  vehiclesez vehiclescash   plaza
         sum          sum   count
0  147065302     54359482  122976
1  185562528     38009405  122976
2  217926485     67000523  122976
3  135142255     21397862  120624
4   46534289      7798630  122976
5   44735980      9676265  122976
6  172800186     26578805  122976
7  106104332     14368223  122784
8  232681943     53530379  122976
9  196120862     38181458   61488
```

DRAFT

Chapter 17

Joins

DRAFT

Contents

1 Helpful Table / Review **291**
2 Merging data in Pandas **292**
3 Complex Join Conditions **294**
4 Stacking Data **294**
5 Lags and Leads **296**
6 Apply, map and applymap: Advanced Transformations **297**

DRAFT

1 Helpful Table / Review

- When we started working in Pandas, we said one of the difficult parts was keeping track of what was being returned by an object. To help with this process, I've created the following, Table 17.1, which maps structure and operation to outcome.
- I personally don't have all these memorized, only a few which allow me to quickly deal with problems.

Data	Operator	Example	Result	Detail
Series	**Aggregation**	`.sum()`	Number	#
	With `.agg`	`.agg('sum')`	Number	#
	With `.agg` in list	`.agg(['sum'])`	Series	Row Index Agg
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	Series	Row Index Col Name
	With `.agg` in dict (lists)	`.agg(['sum', 'count'])`	N/A	Operation not allowed
df	Aggregation	`.sum()`	Series	Row Index Col Name
	With `.agg`	`.agg('sum')`	Series	Row Index Col Name
	With `.agg` in list	`.agg(['sum'])`	df	Row Index Agg
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	Series	Row Index Col Name
	With `.agg` in dict (list)	`.agg({'col1': ['sum']})`	df	Row Index Agg, possible Nulls
groupby	Aggregation	`.sum()`	df	Cols single-level idx
	With `.agg`	`.agg('sum')`	df	Cols single-level idx
	With `.agg` in list	`.agg(['sum'])`	df	Cols multiindex
	With `.agg` in dict (string)	`.agg({'col1' : 'sum'})`	df	Cols single-level idx
	With `.agg` in dict (list)	`.agg({'col1': ['sum']})`	df	Cols multiindex

Table 17.1: Pandas common operations and their results. Bolded are recommended forms.

DRAFT

2 Merging data in Pandas

- To merge DataFrames in Pandas we use the `pd.merge` command.
- The basic structure of merging is the same as in SQL. We need to identify (a) which column(s) we wish to merge on and (b) what type of merge we wish to do.
- There is one wrench that gets thrown into this, however, which is that Pandas requires you to identify if the column(s) you are merging on are part of an index or not.
- In terms of the *type* of merges, they are similar to SQL: left, inner, outer, right and cross are all done the same.
- Let's start with merging two datasets without an index, as demonstrated by the following example:

```
>>> class1 = pd.DataFrame({"sname": ['John', 'Jim', 'Kyle']
    , "grade": ['A', 'A', 'C']})

>>> class2 = pd.DataFrame({"sname": ['John', 'Jim', 'Ashley']
    , "grade": ['A', 'B', 'F']})

>>> pd.merge(class1, class2, on='sname', how='left')
  sname grade_x grade_y
0  John       A       A
1   Jim       A       B
2  Kyle       C     NaN
```

- We call the function using `pd.merge` and then provide it the DataFrames being merged. In this case we provided two DataFrames, class1 and class2. The first DataFrame is considered the "left" DataFrame and the second is considered the "right" DataFrame. We can specify "left" and "right" as parameters if we want to be pedantic `pd.merge(left=class1, right=class2, on='sname', how='l`
- The merge type, `how`, accepts any standard join: left, inner, outer, right and cross as a string.
- We use the `on` parameter to state which column(s) we are merging on. If the columns have the same name then we simply put them in the `on` parameter within the method. If we have more than one column to merge on, we can specify the columns in a list:

```
>>> pd.merge(class1, class2, on=['sname', 'grade'], how='inner')
  sname grade
0  John     A
```

- If the columns are named different things, then we use the "left_on" and "right_on" operators to do the merge:

```
>>> class1T = class1.rename(columns={ 'sname' : 's2'})

>>> pd.merge(class1T, class2, left_on='s2', right_on='sname', how = 'left')
     s2 grade_x sname grade_y
0  John       A  John       A
1   Jim       A   Jim       B
2  Kyle       C   NaN     NaN
```

DRAFT

- One interesting thing that pandas can do is create an indicator which tells you how the row came into the resulting dataset, called `_merge`, as in the example below:

```
>>> pd.merge(class1, class2, on='sname', how='outer', indicator=True)
    sname grade_x grade_y      _merge
0    John       A       A        both
1     Jim       A       B        both
2    Kyle       C     NaN   left_only
3  Ashley     NaN       F  right_only
```

- Instead of calling `merge` directly from the `pandas` module, you can also call it as a method from a DataFrame. When doing this, the calling DataFrame is considered the left DataFrame:

```
>>> class1.merge( class2, how='left', on='sname')
  sname grade_x grade_y
0  John       A       A
1   Jim       A       B
2  Kyle       C     NaN
```

  I prefer to call `pd.merge` rather than using the above notation. I find it a bit cleaner.

- To do a cross-join in pandas (only available in versions greater than 1.5), you state the merge type as `cross` and do not put in any `on` condition.

```
>>> pd.merge(class1, class2, how='cross')
  sname_x grade_x sname_y grade_y
0    John       A    John       A
1    John       A     Jim       B
2    John       A  Ashley       F
3     Jim       A    John       A
4     Jim       A     Jim       B
5     Jim       A  Ashley       F
6    Kyle       C    John       A
7    Kyle       C     Jim       B
8    Kyle       C  Ashley       F
```

- **Index Merging:** In all of the examples above the data which was being merged on was stored as a value and was not a part of the index. If the column being merged on is in an index in one of the DataFrames then instead of using left_on and right_on, the parameters left_index and right_index need to be used, where a boolean True/False is given in the function. Consider the following example:

```
>>> class1idx = class1.set_index('sname')

>>> pd.merge(class1idx, class2, left_index=True, right_on = 'sname', how='left')
    grade_x sname grade_y
0.0       A  John       A
1.0       A   Jim       B
NaN       C  Kyle     NaN
```

  The first command in the above example changes the column "sname" in class1 to an index.

- **BIG THING:** When merging data with pandas *Nulls will match!* This is unlike SQL which has a

consistent treatment of Null values. Per the pandas documentation:

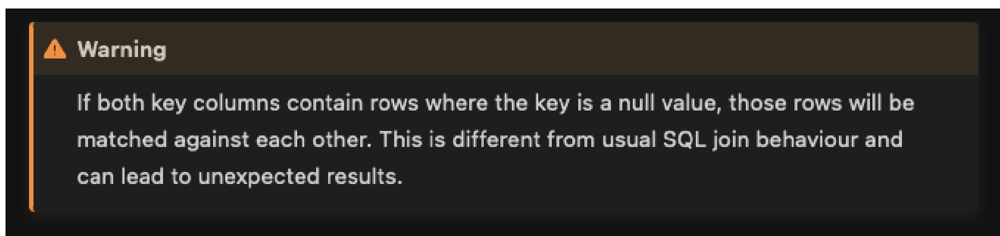


Figure 17.1: Null treatment in merges

3 Complex Join Conditions

- In all of the above examples we had simple equality joins where we wanted to match one column to its exact match within another column in a different DataFrame.
- However, there are many situations where a merge needs to be completed based on a more complex join condition, such as an inequality ($\geq$).
- Pandas, sadly, doesn't provide an easy method to implement non-equality join conditions. This means that when we join, we must either create a cross join style merge and then remove those rows that fail our actual join condition or use an equality join followed by the same filtering method.
- Let's return to our class tables and say that we want to join rows that have names that *don't* match. For example, I wish to create a dataset which allows me to compare each person against everyone else in the same class:

```
>>> d_1 = pd.merge( class1, class1, how='cross')

>>> d_1 = d_1.loc[(d_1.loc[:, 'sname_x'] != d_1.loc[:, 'sname_y']), :]

>>> d_1
  sname_x grade_x sname_y grade_y
1    John       A     Jim       A
2    John       A    Kyle       C
3     Jim       A    John       A
5     Jim       A    Kyle       C
6    Kyle       C    John       A
7    Kyle       C     Jim       A
```

In this case we implemented our more complex join condition *after* we did a cross-join style merge.

4 Stacking Data

- If we have a dataset and wish to stack or append it to another data set (similar to SQL's UNION or UNION ALL) we can use the "concat" operator. This operator takes a DataFrame and then puts multiple copies of the data back-to-back in a specified manner.
- Let's look at the following example:

```
>>> pd.concat([class1,class2])
    sname grade
0    John     A
1     Jim     A
2    Kyle     C
0    John     A
1     Jim     B
2  Ashley     F
```

The concat function, which is in the main pandas library, like merge, takes in data objects and then returns those data objects combined. There are two primary ways that concat is used, the first is above, in which case we wish to stack vertically.

- The concat method can also stack data frames horizontally. For example:

```
>>> pd.concat([class1, class2], axis=1)
  sname grade   sname grade
0  John     A    John     A
1   Jim     A     Jim     B
2  Kyle     C  Ashley     F
```

In the above example, the parameter "axis=1" was added. This parameter tells the concat method to stack the data along columns, rather than along rows. The default behavior is, unsurprisingly, "axis=0" which is the behavior in the previous example.

- A BIG difference between how pandas does concatenation and how relational databases do concatenation is that the columns in pandas are put in **name-alignment**. In other words, only columns which have the same name are matched together. Consider the following example:

```
>>> print("## Note that this is just class2 with a new column name and a new order")
## Note that this is just class2 with a new column name and a new order

>>> class3 = pd.DataFrame({"grade2": ['A', 'B', 'F']
    , "sname": ['John', 'Jim', 'Ashley'] })

>>> pd.concat([class1, class3])
    sname grade grade2
0    John     A    NaN
1     Jim     A    NaN
2    Kyle     C    NaN
0    John   NaN      A
1     Jim   NaN      B
2  Ashley   NaN      F
```

In the example above we see that grade is filled in with "NaN" values for data which was taken from the second DataFrame while grade2 contains "NaN" values for those observations taken from the first DataFrame.

Note also that the columns class1 and class3 were *not* in the same order and the function aligned those columns to those with similar names. In other words, this only appends columns which have the same name.

- One parameter of interest is the parameter "join" which defines which columns to return. If join is set to "inner" then only those columns in both DataFrames are included in the returned DataFrame

while if "outer" is set, all columns are returned. Consider the following examples:

```
>>> class4 = class2.copy()

>>> class4.loc[:, 'test']  = 1

>>> pd.concat([class2, class4], join='inner')
     sname grade
0     John     A
1      Jim     B
2   Ashley     F
0     John     A
1      Jim     B
2   Ashley     F

>>> pd.concat([class2, class4], join='outer')
     sname grade  test
0     John     A   NaN
1      Jim     B   NaN
2   Ashley     F   NaN
0     John     A   1.0
1      Jim     B   1.0
2   Ashley     F   1.0
```

5 Lags and Leads

- A common operation with a DataFrame is to get the previous of next value within a Series. Generally called "lag" and "lead", these operations are done with the `shift` operator, which works on both Series and DataFrames.
- This operator takes in a number which represents how far back (or forward) in the DataFrame to step to get a value.
- Looking at the MTA data set we can use this information to get the previous hour's information:

```
>>> dfMTAC = dfMTA.loc[(dfMTA.loc[:, 'plaza'] == 1) & (dfMTA.loc[:, 'direction'] == 'I'), :]

>>> dfMTAC = dfMTAC.sort_values(['mtadt', 'hr'])

>>> dfMTAC.loc[:, 'pvsCash'] = dfMTAC.loc[:, 'vehiclescash'].shift(1)

>>> dfMTAC.loc[:, 'nxtCash'] = dfMTAC.loc[:, 'vehiclescash'].shift(-1)

>>> dfMTAC.head()
        plaza      mtadt  hr  ... vehiclescash  pvsCash  nxtCash
103440      1 2010-01-01   0  ...          474      NaN    717.0
103442      1 2010-01-01   1  ...          717    474.0    664.0
103444      1 2010-01-01   2  ...          664    717.0    595.0
103446      1 2010-01-01   3  ...          595    664.0    547.0
103448      1 2010-01-01   4  ...          547    595.0    450.0

[5 rows x 8 columns]
```

- In this case the "1" argument in the shift parameter tells Pandas to shift the dataset one row in the forward (or down) direction. In other words, positive values generate lags and negative values

DRAFT

generate leads.

- The order of the rows is set by the `sort_values` command previous in the script. Once the order is set, the shift command steps back a row and the method with the `loc` then sets the values.

- The `shift` operator can also be used in conjunctions with a `groupby` in order to do lags and leads within a particular group. For example:

```
>>> dfMTAC = dfMTA.loc[(dfMTA.loc[:, 'direction'] == 'I'), :]

>>> dfMTAC = dfMTAC.sort_values(['plaza', 'mtadt', 'hr'])

>>> dfMTAgb = dfMTAC.groupby('plaza')

>>> dfMTAC.loc[:, 'pvsCash'] = dfMTAgb.shift(1).loc[:, 'vehiclescash']

>>> dfMTAC.loc[:, 'nxtCash'] = dfMTAgb.shift(-1).loc[:, 'vehiclescash']

>>> dfMTAC.iloc[61487:61490, :]
         plaza      mtadt  hr  ... vehiclescash  pvsCash  nxtCash
1163398      1 2017-01-07  23  ...          191    194.0      NaN
206928       2 2010-01-01   0  ...          290      NaN    363.0
206930       2 2010-01-01   1  ...          363    290.0    346.0

[3 rows x 8 columns]
```

- Note that we created two objects – the copy and one using a `groupby` in order to do this operation. The `groupby` facilitates the segmentation, but to do the assignment we then rely on returning the Series to the copied DataFrame. Since we haven't sorted the data between these operations we can be assured that the rows are still aligned.

- Shift can also work on an entire DataFrame:

```
>>> dfMTAC = (dfMTA
    .loc[ dfMTA.loc[:, 'direction']=='I', ['plaza', 'mtadt', 'hr', 'vehiclesez', 'vehiclescash']]
    .sort_values(['plaza', 'mtadt', 'hr'])
    )

>>> dfMTAC.shift(1).head()
        plaza      mtadt   hr  vehiclesez  vehiclescash
103440    NaN        NaT  NaN         NaN           NaN
103442    1.0 2010-01-01  0.0       415.0         474.0
103444    1.0 2010-01-01  1.0       702.0         717.0
103446    1.0 2010-01-01  2.0       559.0         664.0
103448    1.0 2010-01-01  3.0       480.0         595.0
```

6 Apply, map and applymap: Advanced Transformations

- In this section we consider three advanced methods for transforming columns: `map`, `apply` and `applymap`. These functions allow you to take a DataFrame or Series and apply an arbitrary function to it.

- The first of these we will consider is `applymap` which applies a function to a DataFrame element by element. Note that this function only works on entire DataFrames and not on series:

```
>>> from math import log10

>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head().applymap(log10)
   vehiclescash  vehiclesez
0      2.311754    2.678518
1      2.401401    2.686636
2      2.232996    2.544068
3      2.260071    2.487138
4      2.123852    2.447158
```

- I rarely use the function `applymap` since it applies a function to *every* value in a DataFrame, which isn't that helpful when you have mixed types within a DataFrame *and* that function does not already exist.
- When an alternative exists, `applymap` is generally slower since it applies operations element-by-element rather than vectorizing them in a multi-threaded manner. The two code snippets below do the same operation, but the second is faster (and easier to read).

```
>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head().applymap(lambda x: x**2)
   vehiclescash  vehiclesez
0         42025      227529
1         63504      236196
2         29241      122500
3         33124       94249
4         17689       78400

>>> dfMTA.loc[:, ['vehiclescash', 'vehiclesez']].head() ** 2
   vehiclescash  vehiclesez
0         42025      227529
1         63504      236196
2         29241      122500
3         33124       94249
4         17689       78400
```

- Note also that you pass it a function which takes in a single argument (in the case above this was taking log base 10). If you need to pass in a function which takes in multiple arguments then you will need to use a lambda function.
- The function `map` is the same thing as `applymap` but now on a Series, not on a DataFrame. Once again, it applies an element-by-element operation on a series.

```
>>> dfMTA.loc[:, 'vehiclescash'].head().map(lambda x: x ** 2)
0    42025
1    63504
2    29241
3    33124
4    17689
Name: vehiclescash, dtype: int64
```

- One useful application of `map` is that you can pass it a dictionary and it will apply it as a map to that Series:

DRAFT

```
>>> TransDict =  {1 : 'Robert F. Kennedy Bridge Bronx Plaza (TBX)'
    , 2 : 'Robert F. Kennedy Bridge Manhattan Plaza (TBM)'
    , 3 : 'Bronx-Whitestone Bridge (BWB)'
    , 4 : 'Henry Hudson Bridge (HHB)'
    , 5 : 'Marine Parkway-Gil Hodges Memorial Bridge (MPB)'
    , 6 : 'Cross Bay Veterans Memorial Bridge (CBB)'
    , 7 : 'Queens Midtown Tunnel (QMT)'
    , 8 : 'Brooklyn-Battery Tunnel (BBT)'
    , 9 : 'Throgs Neck Bridge (TNB)'
    , 11 : 'Verrazano-Narrows Bridge (VNB)'}

>>> dfMTA.loc[:, 'plaza'].drop_duplicates().map(TransDict).reset_index(drop=True)
0         Robert F. Kennedy Bridge Bronx Plaza (TBX)
1     Robert F. Kennedy Bridge Manhattan Plaza (TBM)
2                      Bronx-Whitestone Bridge (BWB)
3                          Henry Hudson Bridge (HHB)
4    Marine Parkway-Gil Hodges Memorial Bridge (MPB)
5           Cross Bay Veterans Memorial Bridge (CBB)
6                        Queens Midtown Tunnel (QMT)
7                      Brooklyn-Battery Tunnel (BBT)
8                           Throgs Neck Bridge (TNB)
9                     Verrazano-Narrows Bridge (VNB)
Name: plaza, dtype: object
```

- The last of the complex transforms is `apply` which has both DataFrame and Series methods.
- The reason that `apply` is the most complex is that it is the most general on how it takes in a data as well as what it returns. Consider the following simple examples:

```
>>> d_1 = pd.DataFrame({'A' : [1,2,3], 'B': [4,5,6]})

>>> d_1.apply(np.sum, axis=1)
0    5
1    7
2    9
dtype: int64

>>> d_1.apply(np.sum, axis=0)
A     6
B    15
dtype: int64
```

  In these examples a function is passed to apply which takes in a list and returns a scalar, which is then returned. The axis argument tells tells `apply` in which direction the data is to be passed. When `axis` is equal to 1 then rows are passed to the function while if `axis` is equal to zero, then columns are passed.

- Importantly, `apply` can return complex objects:

```
>>> l_1 = lambda x: pd.Series([sum(x), len(x)])

>>> d_1.apply(l_1, axis=1)
   0  1
0  5  2
1  7  2
2  9  2

>>> d_1.apply(l_1, axis=0)
   A   B
0  6  15
1  3   3
```

The lambda function t returns a Series which is then stacked into a DataFrame by the `apply` method.

- We can also define more complex functions which are specific to the DataFrame in question:

```
>>> def f_1(x): return abs( x.loc['vehiclesez'] - x.loc['vehiclescash'] ) ** 2

>>> dfMTA.head().apply(f_1, axis=1)
0    73984
1    54756
2    32041
3    15625
4    21609
dtype: int64
```

This function does something specific to this DataFrame on a row-by-row basis.

- Note that all three of these methods create a *new* object and that it must be assigned back to the DataFrame if you want to access it later:

```
>>> dfMTAC = dfMTA.copy()

>>> def f_2(x): return abs( x.loc['vehiclesez'] - x.loc['vehiclescash'] ) ** 2

>>> dfMTAC.loc[:, 'newcol'] = dfMTAC.apply(f_2, axis=1)

>>> dfMTAC.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2015-11-28   0         I         477           205   73984
1      1 2015-11-28   0         O         486           252   54756
2      1 2015-11-28   1         I         350           171   32041
3      1 2015-11-28   1         O         307           182   15625
4      1 2015-11-28   2         I         280           133   21609
```

Chapter 18

Window Functions

DRAFT

Contents

1 Window Functions in Pandas **303**
2 Some gotchas **307**
3 Reshaping Data: Transpose, Stack and Unstack **308**
4 A Bunch of stuff to clean up **312**
5 Combining with the original DataFrame **312**
6 Moving the Window **316**
7 Pivot / Melt **316**

DRAFT

1 Window Functions in Pandas

- Pandas has two functions, `expanding` and `rolling` which do SQL style windows aggregations, using a syntax similar to `groupby`.
- An important difference between how Pandas and SQL implement window functions is how sorting is done. In SQL you *never* assume that rows have any order and always apply an ORDER BY clause to sort the data. In Pandas, the sort order is set by operation and you assume that it hasn't change when additional operators are applied. In other words, when we user window functions in SQL we set the row order via the window function but, when we use Pandas, we sort the data ahead of time and assume that the data retains that order.
- The difference between the `rolling` and `expanding` operators is the length of the window under consideration. The `expanding` operator has a window which increases to the start of the DataFrame while the `rolling` operator goes a fixed number of rows behind.
- The `rolling` method has one required parameter, which is the window length. This is similar to setting the `ROWS BETWEEN` operator in SQL.
- The `rolling` method has a fixed window length and, by default, sets all rows which have less data than the window length to NaN.[1]
- Let's consider a simple example to show how this works. We will start by building a simple DataFrame (`df`), which has two columns.

```
>>> d_1 = pd.DataFrame({'c1': [0, 1, 2, np.nan, 4], 'c2' : [0,1,2,3,4]})

>>> d_1
    c1  c2
0  0.0   0
1  1.0   1
2  2.0   2
3  NaN   3
4  4.0   4
```

- Just like `groupby` we use the `rolling` operator on the DataFrame. In this case we are going to choose a window length of two to create a rolling object:

```
>>> x_1 = d_1.rolling(2)

>>> type(x_1)
<class 'pandas.core.window.rolling.Rolling'>
```

- And, just like `groupby` we take this object and apply aggregations to it, using the syntax we have learned before.

  Apply function directly:

[1] This is different than SQL which fills in NULL values when the window length is less than the number of rows.

```
>>> x_1.mean()
     c1   c2
0   NaN  NaN
1   0.5  0.5
2   1.5  1.5
3   NaN  2.5
4   NaN  3.5
```

agg with list:

```
>>> x_1.agg('mean')
     c1   c2
0   NaN  NaN
1   0.5  0.5
2   1.5  1.5
3   NaN  2.5
4   NaN  3.5
```

agg with dict:

```
>>> x_1.agg( {'c1' : ['mean'], 'c2' : ['mean']})
     c1   c2
   mean mean
0   NaN  NaN
1   0.5  0.5
2   1.5  1.5
3   NaN  2.5
4   NaN  3.5
```

Looking at the above, in the first row, both columns have returned NaN. This is because we have set the window size to 2 and, by default, this means that any window of length less than two is set to NaN. We also see that there are two NaN's in the columns `c1`. This is because NaN added to any other number returns NaN.

- We can change the number of observations required to get a response using the `min_periods` argument:

```
>>> d_1.rolling(2, min_periods=1).mean()
     c1   c2
0   0.0  0.0
1   0.5  0.5
2   1.5  1.5
3   2.0  2.5
4   4.0  3.5
```

Note that this changes the result considerably. Since the first row now has a single observation it no longer returns NaN. Surprisingly, even the row with index 3 now has a value since there is one non-NaN value!

- To partition our data, we mix our `rolling` command with the `groupby` operator. In the following command we are going to only look at *inbound* traffic for the sake of simplicity.

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .groupby('plaza')
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
              vehiclescash  vehiclesez
plaza
1     103440           NaN         NaN
      103442           NaN         NaN
      103444        1855.0  558.666667
      103446        1976.0  580.333333
      103448        1806.0  479.000000
```

- Once again, remember that the sort order is set via code and should not be assumed.
- Pandas accumulates distinct values together – even if they are not connected within the original DataFrame. In the second example above, despite the plaza being after the hour column in the sort order this does not mean that multiple plazas are generated per hour. Since the `groupby` is on plaza this means that all similar values, independent of their row order are placed together.
- Take a look at what is returned in the example above and, specifically, what is being returned as the index. Since there was no index in the DataFrame before `rolling` was applied, the command keeps the original `RangeIndex` that was in the DataFrame! This is so that we could merge it back to the DataFrame before the `rolling` command.
- Alternatively, we could have moved our identifying columns into an index *before* specifying the `rolling` command so that we could merge it back onto the original DataFrame:

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
            .sort_values(['plaza', 'mtadt', 'hr'])
            .set_index(['plaza', 'mtadt', 'hr'])
            .groupby('plaza')
            .rolling(3)
            .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                           vehiclescash  vehiclesez
plaza plaza mtadt      hr
1     1     2010-01-01 0            NaN         NaN
                       1            NaN         NaN
                       2         1855.0  558.666667
                       3         1976.0  580.333333
                       4         1806.0  479.000000
```

which would yield two `plaza` columns. However, just turning off `as_index` in the `groupby` won't change this issue:

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .set_index(['plaza', 'mtadt', 'hr'])
          .groupby('plaza', as_index=False)
          .rolling(3)
          .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                            vehiclescash  vehiclesez
plaza plaza mtadt      hr
1     1     2010-01-01 0             NaN         NaN
                       1             NaN         NaN
                       2          1855.0  558.666667
                       3          1976.0  580.333333
                       4          1806.0  479.000000
```

Instead you need to have your index set to the returning variables you care about. Note that the as_index has no effect on what gets returned in this situation, as the rolling command will put plaza into the index no matter what.

```
>>> res = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), :]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .set_index(['mtadt', 'hr'])
          .groupby('plaza', as_index=False)
          .rolling(3)
          .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'}))

>>> res.head()
                      vehiclescash  vehiclesez
plaza mtadt      hr
1     2010-01-01 0             NaN         NaN
                 1             NaN         NaN
                 2          1855.0  558.666667
                 3          1976.0  580.333333
                 4          1806.0  479.000000
```

- So what have we learned:
  - The rolling command will take whatever is in the index and pass it through to the resultant DataFrame.
  - The rolling command will add whatever groupby column appears *as an index*, no matter what options you put in the groupby function.
  - Make sure that your DataFrame is *sorted* before applying the rolling operation.
- The other command used when doing window functions is the expanding operator. This operator calculates the aggregation back to the beginning of the frame in question, rather than based on a fixed window size.
- For example, if we want to return a running sum we could do the following and, we could verify that the changeover happens correctly:

```
>>> d_1 = (dfMTA
          .sort_values(['plaza', 'mtadt', 'hr'])
          .groupby('plaza')
          .expanding().agg({'vehiclesez' : 'sum'})
         )

>>> d_1.iloc[122975:122980]
               vehiclesez
plaza
1     1163399  147065302.0
2     206928         457.0
      206929         986.0
      206930        1526.0
      206931        2273.0
```

2 Some gotchas

Adding Back

- These types of functions are *very* easy to use in ways that cause problems.
- The biggest reason for this is that to run these commands the indexes have to be set *just right*.
- After running these commands we then want to put this data back into our original DataFrame, but this means then either changing the original DataFrame to conform with the result of our operation OR changing the result of our operation so that it conforms to our original DataFrame.
- In either case it is easy to end up in a place where functions do not return an error – but also aren't doing exactly what you want. The commands below are one way that we can take a DataFrame, do our aggregation functions and then add them back to our original DataFrame. Note the complexity required to make sure that the indexes align properly.

DRAFT

```
>>> d_1 = (dfMTA
            .set_index(['plaza', 'mtadt', 'hr', 'direction'])
        )

>>> d_2 = (d_1
        .reset_index(['plaza', 'direction'])
        .sort_values(['plaza', 'mtadt', 'hr'])
        .groupby(['plaza', 'direction'])
        .rolling(3)
        .agg({'vehiclescash' : 'sum', 'vehiclesez' : 'mean'})
        .reset_index()
        .set_index(['plaza', 'mtadt', 'hr', 'direction'])
        ).copy()

>>> d_1.loc[:, 'rcash'] = d_2.loc[:, 'vehiclescash' ]

>>> d_1.loc[:, 'rez'] = d_2.loc[:, 'vehiclesez' ]

>>> d_1.head()
                               vehiclesez  vehiclescash  rcash         rez
plaza mtadt      hr direction
1     2015-11-28 0  I                 477           205  817.0  653.333333
                    O                 486           252  998.0  694.333333
                 1  I                 350           171  646.0  499.666667
                    O                 307           182  797.0  509.333333
                 2  I                 280           133  509.0  369.000000
```

Offsetting

- There are no options within `rolling` or `expanding` to offset the data in some way.
- To do this we have to use the `shift` operator.

3 Reshaping Data: Transpose, Stack and Unstack

- In this section we look at the three commonly used commands for reshaping data between wide- and long-formats: `transpose`, `stack` and `unstack`.
- These operations strongly rely on indexes on both rows and columns. My common workflow with these operations is:

  1. Realize that I need to reshape the data.
  2. Figure out what index I need.
  3. Create index.
  4. Reshape data.
  5. Drop the index.

  I don't use indexes that much, preferring to leave the data "raw", rather than in named index columns. Because of this pattern, when I do need to reshape I have to define the appropriate indexes. This is a bit backward, but my preference is to avoid the complexity of indexes.

DRAFT

- In the simplest case to reshape data we can simply "transpose" it using the operator `T`. Let's look at the following example:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & (dfMTA.loc[:, 'direction'] == 'I')
    & (dfMTA.loc[:, 'plaza']==1),
    ['hr', 'vehiclesez', 'vehiclescash']]
    .reset_index(drop=True))

>>> d_1.head()
   hr  vehiclesez  vehiclescash
0   0         669           315
1   1        1085           426
2   2         922           426
3   3         767           450
4   4         724           429
```

We have three columns of data and we wish to make it wide. There are two options for this data: one is that we have "hr" as a column index or we just have "hr" as a row. We can do either by choosing to set an index or not:

1. **Pure Transpose:** Swap everything.

```
>>> d_1.T
               0     1    2    3    4    5  ...    18    19   20   21   22   23
hr             0     1    2    3    4    5  ...    18    19   20   21   22   23
vehiclesez   669  1085  922  767  724  616  ...  1098  1107  971  844  783  626
vehiclescash 315   426  426  450  429  331  ...   489   482  378  369  344  283

[3 rows x 24 columns]
```

2. **Transpose with an Index:** Create a column index based on hour.

```
>>> d_1.set_index('hr').T
hr             0     1    2    3    4    5  ...    18    19   20   21   22   23
vehiclesez   669  1085  922  767  724  616  ...  1098  1107  971  844  783  626
vehiclescash 315   426  426  450  429  331  ...   489   482  378  369  344  283

[2 rows x 24 columns]
```

   Looking at the result there are only two rows this time since "hr" has been turned into a column index.

- To swap the data back to the original form use the `T` command again.
- Transpose works when you wish to reshape the *entire* DataFrame. Most of the time, however, that operation is too severe and you only wish to make some of the information change shape.
- The first command `stack` takes data which is "wide" and makes it long while `unstack` returns the data to its wide format. Let's look at an example, using the MTA data:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & (dfMTA.loc[:, 'direction'] == 'I')
    & ((dfMTA.loc[:, 'plaza']==1) | (dfMTA.loc[:, 'plaza'] == 2)),
    ['plaza', 'hr', 'vehiclesez', 'vehiclescash']]
    .reset_index(drop=True)
    .set_index(['plaza', 'hr'])
    .unstack('plaza')
    )

>>> d_1.head()
      vehiclesez      vehiclescash
plaza          1    2            1    2
hr
0            669  554          315  160
1           1085  799          426  259
2            922  670          426  320
3            767  518          450  187
4            724  423          429  180
```

- We created a dataset with four columns: plaza, hr, vehiclesez and vehiclescash. We then use unstack to take this "long" data and turn it "wide" along the plaza dimension. The resulting DataFrame will have 24 rows and four columns.
- We can undo this command by using `stack`:

```
>>> d_1.stack('plaza').head()
          vehiclesez  vehiclescash
hr plaza
0  1             669           315
   2             554           160
1  1            1085           426
   2             799           259
2  1             922           426
```

As you can see we have moved plaza from the column index back as a row index. The only difference between this and the original DataFrame is the order of the index, which we could remove with `reset_index`.

- This might seem like magic, but lets think through the operation a bit and see if we can make sense of it. First, when we stack a DataFrame all columns with the same values are treated the same in the resulting DataFrame. This makes the reshape that much easier to conceptualize: all examples of plazas with the same number are going to have the number when we stack.
- The `unstack` operation also only works if the index that is set is **unique for each row.** By doing this, there is no way to have a conflict on the reshape.
- If we make the data wide by `unstack`, there may not be values present in all varieties of each index value. The `stack` operation, on the other hand, does not create any new data, so missing values won't be created.
- To use these operations its important to consider the following:
  - What values do you want in the new rows and columns: Are they unique? If not, stop.

  - Once you have identified which values are moving, determine what is a value and what should be in the index.
  - Set the index
  - Call `stack` or `unstack` with the appropriate variable, from the index, selected.

- Note that you can do multiple values in your reshaping by providing a list. Consider the following:

```
>>> d_1 = (dfMTA.loc[(dfMTA.mtadt == '2016-01-01')
    & ((dfMTA.loc[:, 'plaza']==1) | (dfMTA.loc[:, 'plaza'] == 2)),
    ['plaza', 'hr', 'vehiclesez', 'direction', 'vehiclescash']]
    .reset_index(drop=True)
    . set_index(['plaza', 'hr', 'direction'])
    .unstack(['plaza', 'direction'])
    )

>>> d_1.head()
          vehiclesez                      vehiclescash
plaza              1         2                       1         2
direction          I    O    I     O                 I    O    I    O
hr
0                669  552  554   760               315  300  160  241
1               1085  896  799  1123               426  437  259  357
2                922  747  670   933               426  447  320  360
3                767  694  518   728               450  407  187  257
4                724  577  423   586               429  369  180  188
```

- Returning to the above, we can also do a "semi" stack:

```
>>> d_1.stack('plaza').head()
          vehiclescash      vehiclesez
direction            I    O          I     O
hr plaza
0  1               315  300        669   552
   2               160  241        554   760
1  1               426  437       1085   896
   2               259  357        799  1123
2  1               426  447        922   747
```

4 A Bunch of stuff to clean up

- You can see this in the below (no idea what we are talking about)
- When using `expanding` or `rolling` keep in mind that the DataFrame returned does *not* have a clean index system. Continuing with the above example:

```
>>> d_1.index.names
['plaza', 'mtadt', 'hr', 'direction']
```

Unexpected! There are two levels of the index: one generated from the plaza `groupby` and another with the name "None". Even if we decide to stop the index creation with the `groupby` we will end up with an unexpected result:

```
>>> d_2 = (dfMTA
    .sort_values(['plaza', 'mtadt', 'hr'])
    .groupby('plaza', as_index=False)
    .expanding().agg({'vehiclesez' : 'sum'})
    )

>>> d_2.index.names
['plaza', None]
```

Comparing the above, we see that both, dfMTAC and dfMTAC2 have an additional index column:

```
>>> d_1.head()
                                vehiclesez  vehiclescash  rcash         rez
plaza mtadt      hr direction
1     2015-11-28 0  I                  477           205  817.0  653.333333
                    O                  486           252  998.0  694.333333
                 1  I                  350           171  646.0  499.666667
                    O                  307           182  797.0  509.333333
                 2  I                  280           133  509.0  369.000000

>>> d_2.head()
              vehiclesez
plaza
1     103440       415.0
      103441       801.0
      103442      1503.0
      103443      2037.0
      103444      2596.0
```

- This additional index column has implications for how we combine this data with other DataFrames, as we will see below.

5 Combining with the original DataFrame

- In the previous examples we generated a new Series or DataFrame which contained the data that we were interested in. Frequently we wish to combine this new data with the DataFrame that generated it and, sadly, this can be difficult as we need to create the column and then somehow put it back on

the original dataset.[2]

- There are a few different possibilities when doing this:
  1. `rolling` or `expanding` without a `groupby`.
  2. `rolling` or `expanding` with a `groupby` by creating an index.
  3. `rolling` or `expanding` with a `groupby` by using an already present index.

  We will go over each in the section below.

Without a groupby

- When there is no `groupby` we simply compute the `expanding` or `rolling` values, reset the index and then select the column and join back on:

```
>>> d_1 = dfMTA.copy()

>>> d_1.loc[:, 'newcol'] = (d_1
                                    .expanding()
                                    .agg({'vehiclescash' : 'sum'})
                                    .reset_index()
                                    .loc[:, 'vehiclescash'])

>>> d_1.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2015-11-28   0         I         477           205   205.0
1      1 2015-11-28   0         O         486           252   457.0
2      1 2015-11-28   1         I         350           171   628.0
3      1 2015-11-28   1         O         307           182   810.0
4      1 2015-11-28   2         I         280           133   943.0
```

- In the case where we want to sort the data beforehand, it is import to `sort_values` as well as `reset_index` on the original DataFrame to make sure that everything stays aligned:

```
>>> d_1 = dfMTA.sort_values(['mtadt', 'hr']).reset_index(drop=True).copy()

>>> d_1.loc[:, 'newcol'] = (d_1
                .expanding()
                .agg({'vehiclescash' : 'sum'})
                .reset_index()
                .loc[:, 'vehiclescash'])

>>> d_1.head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash  newcol
0      1 2010-01-01   0         I         415           474   474.0
1      1 2010-01-01   0         O         386           412   886.0
2      2 2010-01-01   0         I         457           290  1176.0
3      2 2010-01-01   0         O         529           321  1497.0
4      3 2010-01-01   0         I         701           406  1903.0
```

[2] I'm really open to being wrong on this, but after spending a significant amount of time on this, I haven't seen a consistent solution outside what is shown here.

Note that the only difference between the two previous code blocks is the `sort_values` and `reset_index` commands.

With a GroupBy and Creating an Index

- Let's say that we don't have an obvious set of index columns to use, but we still wish to use a `groupby` with a window function. In this case we need to create an index.
- Consider the following situation where we want to calculate the running sum of inbound cars over the entire DataFrame, but partitioned by plaza:

```
>>> d_1 = (dfMTA
          .loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .reset_index(drop=True)
         )

>>> d_1.index
RangeIndex(start=0, stop=613608, step=1)
```

At this stage we have set up our original dataset to be sorted correctly and created a new integer index. The reason for the `drop=True` line is to prevent the original index from being placed in the DataFrame.[3]

We now take this DataFrame and create our running sum, making sure to start from the sorted DataFrame:

```
>>> d_2 = (d_1
          .groupby('plaza', sort=False)
          .expanding()
          .agg({'vehiclescash' : 'sum'})
          )

>>> d_2.head()
         vehiclescash
plaza
1     0         474.0
      1        1191.0
      2        1855.0
      3        2450.0
      4        2997.0

>>> d_2.index.names
['plaza', None]
```

Looking at the above, we can see that the index is no longer a `RangeIndex` and has changed! Meaning that we probably can't merge it back onto the original DataFrame without some modification.

- Note also that we included the option "sort=False" in our `GroupBy`. We did this because we want to make sure that this method doesn't change the order of the data. Since we know that the order is going to be stable, we reset the index:

[3]The original index was also an RangeIndex, but since we dropped all of the outbound rows as well as sorted the DataFrame, the original index does not exist in the proper form.

```
>>> d_2.loc[ :, 'runningsum'] = d_2.reset_index().loc[:, 'vehiclescash']

>>> d_2.head()
         vehiclescash  runningsum
plaza
1     0         474.0         NaN
      1        1191.0         NaN
      2        1855.0         NaN
      3        2450.0         NaN
      4        2997.0         NaN
```

- Combining this all together into two lines:

```
>>> d_1 = (dfMTA
          .loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .reset_index(drop=True)
         )

>>> d_1['runningsum'] = (d_1
                        .groupby('plaza', sort=False)
                        .expanding()
                        .agg({'vehiclescash' : 'sum'})
                        .reset_index(drop=True)
                        .loc[:, 'vehiclescash']
                       )
```

With a GroupBy using an index

- Alternatively, we can rely on an unique set of index column if they are present in the DataFrame. Redoing the example above:

```
>>> d_1 = (dfMTA.loc[ (dfMTA.loc[:, 'direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .set_index(['plaza', 'mtadt', 'hr'])
         )

>>> d_1.loc[:, 'runningsum'] = (d_1
          .reset_index('plaza')
          .groupby('plaza', as_index=False, sort=False)
          .expanding()
          .agg({'vehiclescash' : 'sum'})
          .reset_index()
          .set_index(['plaza', 'mtadt', 'hr'])
          .loc[:, 'vehiclescash'])
```

- Looking at the above, we `set_index` on the original DataFrame and then set it again on the created dataset.

- IMPORTANT: A caveat to the above is that if the index columns are not unique then we can run into situations where the data is sorted differently in each and thus the merge may result in incorrect results. This method should *only* be used if there is a set of columns which uniquely define a row.

6 Moving the Window

- A limitation in how Pandas implement window functions is that they do not naturally have the ability to move the window – e.g. offset it by a number of rows.
- For example, lets say that I want to know the maximum value of a column up to, but not including the current row? This could occur because I want to know if the current row is higher than the previous maximum value. It's easy enough to calculate the maximum up to, and including the current row, but moving that window back one requires an additional operation.
- One way of doing this is to use the `shift` operator to move the data after the calculation occurs, such as in the example below which calculates the maximum vehicles which use an cash up to, but not including the current row (only in the inbound direction)

```
>>> d_1 = (dfMTA.loc[ (dfMTA['direction'] == 'I'), ['plaza', 'mtadt', 'hr', 'vehiclescash']]
          .sort_values(['plaza', 'mtadt', 'hr'])
          .set_index(['plaza', 'mtadt', 'hr'])
         )

>>> d_1.loc[:, 'runningmax_no_current'] = (d_1
          .reset_index('plaza')
          .groupby('plaza', as_index=False, sort=False)
          .expanding()
          .agg({'vehiclescash' : 'max'})
          .reset_index()
          .set_index(['plaza', 'mtadt', 'hr'])
          .groupby('plaza', as_index=False)
          .shift(1)
          .loc[:, 'vehiclescash']
)

>>> d_1 = d_1.reset_index()
```

- The last line removes the index that we created.

7 Pivot / Melt

- While we won't cover it in this course, the `pivot` and `melt` commands are powerful way to reshape data.
- While they nearly map to `stack` and `unstack`, they do not require the use of an index.

DRAFT

Appendix A

Data Dictionaries

DRAFT

1 Introduction

This chapter contains information on the data used in this course and how to load it into PostgreSQL and Pandas. To begin loading the data, clone the git repo that can be found at `https://github.com/NickRoss/sql-data`.

The repo itself contains a script (`load_data.py`) which will load the data into a PostgreSQL compatible database as well as a Docker image for running PostgreSQL via containers. If one wishes to load the data themselves, this appendix contains a basic framework for loading each table.

This section also contains information on how to load some of the datasets into Pandas. In both cases (Pandas and SQL), the <FILEPATH> parameter needs to be changed to the location of the file on your local machine.

2 Iowa Fleet data

This table contains automobile registration information and annual fees for the state of Iowa. Note that a few changes were made to the file. In particular, O'brien county was miscoded at times and NULL counties were removed. The final table contains 41,202 rows of data.

CREATE TABLE and COPY commands which load the data into a PostgreSQL compatible database can be found below:

```
create table cls.cars (
    year int
    , countyname varchar(20)
    , motorvehicle varchar(3)
    , vehiclecat varchar(15)
    , vehicletype varchar(55)
    , tonnage  varchar(30)
    , registrations int
    , annualfee float
    , completecategory varchar(90)
);

COPY cls.cars FROM  '<FILEPATH>/iowa_cars.tdf'
        CSV DELIMITER AS E'\t'
```

To load the data into Pandas, the following command can be used:

```
dfCars = pd.read_csv('<FILEPATH>/iowa_cars.tdf',
                 sep='\t', engine='python', names=['year', 'countyname',
                 'motorvehicle' ,'vehiclecat', 'vehicletype',
                 'tonnage', 'registrations', 'annualfee',
                 'completecategory'])
```

Table A.1: Data Dictionary for Iowa Cars Data

Column	Example Values	Data Type	Description
Year	2011	Int	Calendar year vehicle was registered
CountyName	“Adair”	Varchar(20)	County vehicle was registered. Those without a county listed were registered/titled by the State
MotorVehicle	“Yes”	VarChar(3)	Indicates whether motor vehicle (Yes) or trailer (No).
VehicleCat	“Trailer”	VarChar(15)	Broad category for vehicle types.
Vehicletype	“Bus”	VarcChar(25)	Type of vehicle registered.
tonnage	“4 tons”	charchar(30)	Tonnage category for truck and truck tractor vehicle types.
registrations	397	int	Number of vehicle registrations.
annualfee	1470	float	Annual fee associated with vehicle registrations.
completecategory	“Truck – 3 Tons”	varchar(90)	Combination of VehicleType and tonnage.

3 NY MTA Data

The data in this table represents hourly traffic on NY's MTA system.[1] Information in Table A.1 contains the map between toll plaza ID and the name of the toll plaza. The final table contains 1,165,728 rows of data.

In order to load the data use the following set of commands:

```
create table cls.mta (
                plaza int
                , mtadt date
                , hr int
                , direction varchar(1)
                , vehiclesEZ int
                , vehiclesCASH int
);

COPY cls.mta from '<FILEPATH>/MTA_Hourly.tdf'
        CSV DELIMITER AS E'\t'
```

Loading the data into Pandas can be accomplished with the following command:

[1] Information was downloaded from this location: https://catalog.data.gov/dataset/hourly-traffic-on-metropolitan-transportation-authority-mta-bridges-and-tunnels-beginning-

```
dfMTA = pd.read_csv('<FILEPATH>/MTA_Hourly.tdf',
                  sep='\t', engine='python', names=['plaza', 'mtadt',
                  'hr', 'direction', 'vehiclesez', 'vehiclescash'])

dfMTA.mtadt = pd.to_datetime(dfMTA.mtadt)
```

Column	Example Values	Data Type	Description
plaza	1,2,3	Int	Plaza Number (more information below)
mtadt	1/1/2012	Date	Observation Date
hr	0 - 23	Int	Hour of observation
direction	I, O	varchar(1)	Direction of traffic (Inbound vs. Outbound)
vehiclesez	1254	int	The number of vehicles that pass through each bridge and pay with EZ pass
vehiclescash	1254	int	The number of vehicles that pass through each bridge and pay with cash

Figure A.1: Information on Plaza number for MTA Hourly data

Plaza ID	Name
1	Robert F. Kennedy Bridge Bronx Plaza (TBX)
2	Robert F. Kennedy Bridge Manhattan Plaza (TBM)
3	Bronx-Whitestone Bridge (BWB)
4	Henry Hudson Bridge (HHB)
5	Marine Parkway-Gil Hodges Memorial Bridge (MPB)
6	Cross Bay Veterans Memorial Bridge (CBB)
7	Queens Midtown Tunnel (QMT)
8	Hugh L. Carey Tunnel (HLC) formally known as Brooklyn-Battery Tunnel (BBT)
9	Throgs Neck Bridge (TNB)
11	Verrazano-Narrows Bridge (VNB)

DRAFT

4 Daily Stock Data: s2010 and s2011

The tables s2010 and s2011 contain information on daily prices for stocks that appear on the NYSE or NASDAQ. The table s2010 has 816,066 rows while the table s2011 has 864,110 rows.

The columns **symb and retdate** define a unique row for each table.

The commands below will generate two tables, s2010 and s2011 in the schema "stocks" and then load the data into those two tables. Note that "<FILEPATH>" has to be changed to the path of where the data lies in on the machine which is loading the data.

```
create table stocks.s2010 (
        symb varchar(6)
        , retdate date
        , opn float
        , high float
        , low  float
        , cls float
        , vol int
        , exch varchar(8));

COPY stocks.s2010 FROM
         '<FILEPATH>/s2010.tdf'
        CSV DELIMITER E'\t';

reate table stocks.s2011 (
        symb varchar(6)
        , retdate date
        , opn float
        , high float
        , low  float
        , cls float
        , vol int
        , exch varchar(8));

COPY stocks.s2011 FROM '<FILEPATH>/s2011.tdf'
        CSV DELIMITER E'\t';
```

To load the data into Pandas DataFrames, use the following command:

```
df2010 = pd.read_csv('<FILEPATH>/s2010.tdf',
                 sep='\t', engine='python',
                 names=['symb', 'retdate', 'opn',
                 'high', 'low', 'cls', 'vol', 'exch'])



df2011 = pd.read_csv('<FILEPATH>/s2011.tdf',
                 sep='\t', engine='python',
                 names=['symb', 'retdate', 'opn',
                 'high', 'low', 'cls', 'vol', 'exch'])
```

If you wish to have the dates be converted to dates you can use the following commands to update the DataFrame.

```
df2010[:, 'retdate'] = pd.to_datetime(df2010.retdate)
df2011[:, 'retdate'] = pd.to_datetime(df2011.retdate)
```

Alternatively, you can load `retdate` as a date using the following:

```
df2010D = pd.read_csv('../sql-data/raw_data/s2010.tdf'
                 ,sep='\t', engine='python', names=['symb'
                 , 'retdate','opn', 'high', 'low', 'cls',
                 'vol', 'exch'], parse_dates = ['retdate'])

df2011D = pd.read_csv('../sql-data/raw_data/s2011.tdf'
                 ,sep='\t', engine='python', names=['symb'
                 , 'retdate','opn', 'high', 'low', 'cls',
                 'vol', 'exch'], parse_dates = ['retdate'])
```

Column	Type	Description
symb	Varchar	Code for the stock being traded.
retdate	Date	Date for the stock being traded.
opn	float	The open price of the stock.
high	float	The high price of the stock that day.
low	float	the low price of the stock that day.
cls	float	the closing price of the stock that day.
vol	int	the number of share traded that day.
exch	varchar	what exchange the stock is traded on.

5 Annual Fundamental Financial information: fnd

The tables fnd contains information taken from annual reports for stocks. The key to these tables are the columns datadate and gvkey. The table has 33,817 rows of data and spans most of 2010 and 2011.

The commands below will load the fnd data in the schema “stocks”. Note that “<FILEPATH>” has to be changed to the path of where the data lies in on the machine which is loading the data.

DRAFT

```
create table stocks.fnd (
        gvkey varchar(8)
        , datadate date
        , fyear int
        , indfmr varchar(4)
        , consol varchar(1)
        , popsrc varchar(1)
        , datafmt varchar(3)
        , tic varchar(8)
        , cusip varchar(11)
        , conm varchar(30)
        , fyr int
        , cash float
        , dp float
        , ebitda float
        , emp float
        , invt float
        , netinc float
        , ppent float
        , rev float
        , ui float
        , cik varchar(10)
    );

COPY stocks.fnd FROM  '<FILPATH>/fnd.tdf
        CSV DELIMITER E'\t';
```

To load the data into Pandas, use the following command:

```
dffnd = pd.read_csv('<FILEPATH>/fnd.tdf'
                ,sep='\t', engine='python', names=['gvkey', 'datadate',
                      'fyear', 'indfmr', 'consol', 'popsrc', 'datafmt', 'tic'
                      , 'cusip', 'conm', 'fyr', 'cash', 'dp', 'ebitda', 'emp'
                      , 'invt' , 'netinc', 'ppent', 'rev', 'ui', 'cik'])
```

Column	Min. Val/Len	Max. Val/Len	Description
cash	-0.01	168896.51	The amount of cash on the balance sheet. Measured in millions of dollars.
cik	10	10	SEC identifier for corporations.
conm	3	30	Company Name
consol	1	1	If the information is consolidated with subsidiaries or kept separate.
cusip	9	9	Another identifier, this one maintained by the CUSIP bureau.
datafmt	3	3	Represents how the data was collected.
dp	-0.24	23713.56	GAAP depreciation and Amortization from the income statement. Measured in Millions of dollars.
ebitda	-45026.00	124840.00	Earnings Before Interest Taxes and Depreciation, measured in millions of dollars
emp	0.00	2100.00	Number of employees, measured in thousands.
fyear	2008	2011	Fiscal year. Note that a fiscal year is defined as the year with the most months of the calendar year with June falling forward.
fyr	1	12	Month in which the fiscal year ends.

DRAFT

Column	Min. Val/Len	Max. Val/Len	Description
gvkey	6	6	Unique Company Identifier used in the Fundamental Data
indfmr	4	4	Represents how the information is presented in the database.
invt	0.00	373176.43	Inventory from the balance sheet. Measured in Millions of Dollars.
netinc	-71969.00	104821.00	Net Income, in millions of dollars from the Income Statement.
popsrc	1	1	Source of the data. D means Domestic.
ppent	0.00	218567.00	Total Property Plants and Equipment from the Balance Sheet, measured in Millions of Dollars.
retdate	8	8	Data Date: Date which the information becomes available to the public. Represents the date of the fiscal year-end.
rev	-6749.63	470171.00	Total Sales from the Income Statement, measured in Millions of Dollars
tic	1	8	Ticker Symbol. Note that this is modified under certain circumstances.
ui	0.00	0.00	Unearned Income, measured in millions of dollars.

6 Soap Transaction Data

This table consists of information relating to a subscription soap service. There are two ways that customers can order: either via subscription or by a one-off ("unit") purchase. There are two different order types: single bars or double bars, though an order can have multiple of a single type in it. For example, if a row is double bars and there are "2" in the units column, this means that there were four total bars in the order associated with that row. The table has 1,047,381 rows of data.

The following commands define a table for the soap data as well as populate that table.

DRAFT

```
create table cls.trans (
        orderid int
        , userid int
        , trans varchar(15)
        , type varchar(15)
        , local varchar(10)
        , trans_dt date
        , units int
        , coupon float
        , months int
        , amt float );

COPY cls.trans from '<FILEPATH>/soapData.tdf'
        CSV DELIMITER E'\t';
```

Column	Example Values	Data Type	Description
orderid	1,2,3	Int	Unique ID for the order
userid	1,2,3	int	Unique ID for the user
trans	Double Bar	varchar	Bar type in order
type	Unit, Sub	varchar	Is this part of a subscription or one off transaction?
local	Mexico	varchar	Location of the customer
trans_dt	date	12/22/2016	Date of the transaction
units	1,2,3	int	Number of that trans in the order
coupon	.25	float	the percent coupon applied
months	1,2,3	int	If a subscription, the timing of the subscription
amt	47.96	float	The total price of the transaction

To load the data into Pandas, use the following command:

```
dfTrans = pd.read_csv('<FILEPATH>/soapData.tdf'
                     , sep='\t', engine='python', names = ['orderid',
                     'userid', 'trans', 'type', 'local', 'trans_dt',
                     'units', 'coupon', 'months', 'amt'],
                     parse_dates=['trans_dt'])
```

DRAFT

Appendix B

Connecting SQL to Python or R

In this Appendix we will briefly look at how to connect SQL to Python or R. Unfortunately connecting both programming languages can be difficult. There are a number of different ways to make the connection.

1 Connecting to any database: ODBC and JDBC

ODBC is an API for connecting to database systems. It was originally developed in the early 1990's, though it is still in use today. RODBC and PyODBC are the primary ways of connecting R and Python to ODBC interfaces.

JDBC is a JAVA extension of ODBC. It is the primary way that most SQL clients connect to databases. In order to use JDBC, a JDBC driver for a database must be provided. In the case of PopSQL, the driver is built-in, though most other SQL clients require the drivr to be downloaded.

JayDeBeApi and RJDBC are the two common tools for connecting Python and R to databases via JDBC drivers.

In both cases (ODBC and JDBC), the driver provides a standard interface between the client application and the database. These interfaces are specific to the database – you can think of them as printer drivers for your database. The PostgreSQL JDBC driver will not allow you to connect to a MS-SQL database.

2 Connecting only to PostgreSQL

If you only wish to connect to a single database variant, then you can use packages and programs that are built around that server, rather than more generalized packages.

Generally speaking variant specific tools tend to be more robust and easier to use. The downside is that information about one cannot necessarily be used for other variants.

For Python, the most common tool for connecting to PostgreSQL is the package `psycopg2` while R has a package `RPostgreSQL`. Using either tool requires setting up a connect and then sending queries through that connection.

As an example, the following Python code attempts to create a table. If there is an error, the connection is reset. Note the use of both the "commit" command and the "rollback" command. These are necessary because `psycopg2` does not automatically commit its transaction. Keep in mind that the code below will not run without providing host, db name, user and password information.

Installing `psycopg2` can be difficult. On Macs I install it using brew, though it can be installed via other library management tools.

DRAFT

```
conn_string = "host='%s' dbname='%s' user='%s' password='%s'" % (ahost, adbname, aUser, apass)
Sconn = psycopg2.connect(conn_string)
Scur = Sconn.cursor()

cmds = [ """create table cls.cars (
    year int
    , countyname varchar(20)
    , motorvehicle varchar(3)
    , vehiclecat varchar(15)
    , vehicletype varchar(55)
    , tonnage  varchar(30)
    , registrations int
    , annualfee float
    , completecategory varchar(90)
);"""]

for x in cmds:
        try:
                Scur.execute(x)
                Sconn.commit()
        except psycopg2.ProgrammingError:
                print( """CAUTION FAILED: '%s' """ % x)
                Sconn.rollback()
```

Using RPostgreSQL, the code below will create an object with the result of the query:

```
require("RPostgreSQL")
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "XXX", host = "localhost"
                , port = 5432, user = "XXX", password = "XXX")
df_postgres <- dbGetQuery(con, "SELECT * from cls.traffic;")
```

DRAFT

Appendix C

Assignments

DRAFT

1 HW #0A: PostgreSQL Installation

Complete the tasks below in order to complete the assignment. Importantly, nothing needs to be turned in to complete this assignment.

Nothing needs to be physically turned in for this assignment, just make sure to complete the final step.

1. Install PostgreSQL on your computer. Note that installing PostgreSQL can be difficult and I recommend doing some research before beginning. If you are using a mac, I recommend using homebrew to install it. There is also something called PostgresAPP, which you can try.[1]

2. Once PostgresSQL is installed, please create a schema for the stocks database, which can be done using the command below.

   ```
   create schema stocks;
   commit;
   ```

   We will also create a schema for some other datasets that are using in the class, which is "cls" and can be done using the commands below:

   ```
   create schema cls;
   commit;
   ```

3. All of the data required for the homework can be found on the canvas page and the queries required to load the data onto your Postgres instance can be found in the data dictionary. Broadly speaking to load the data you must:

   (a) Have a schema to place the table in (which is what was done in the previous step)

   (b) Create a table to load the data into (the CREATE TABLE commands can be found in the data dictionary)

   (c) Use a COPY command to move the data from its raw format into the database.

4. Please load the following datasets onto your local SQL instance: (1) stocks.s2010 (2) stocks.s2011, (3) stocks.fnd

5. For an SQL Client, I would recommend using PopSQL. One important trick when installing is that if you are referring to your local machine the host is "localhost."

6. Make sure that on Slack and on the Canvas page you have a photo of yourself that will help me recognize you. After bootcamp you are free to make your Slack icon whatever you want, but during bootcamp you must have a recognizable photo as your avatar.

DRAFT

[1] I am not IT and am not going to diagnose issues relating to installing your software.

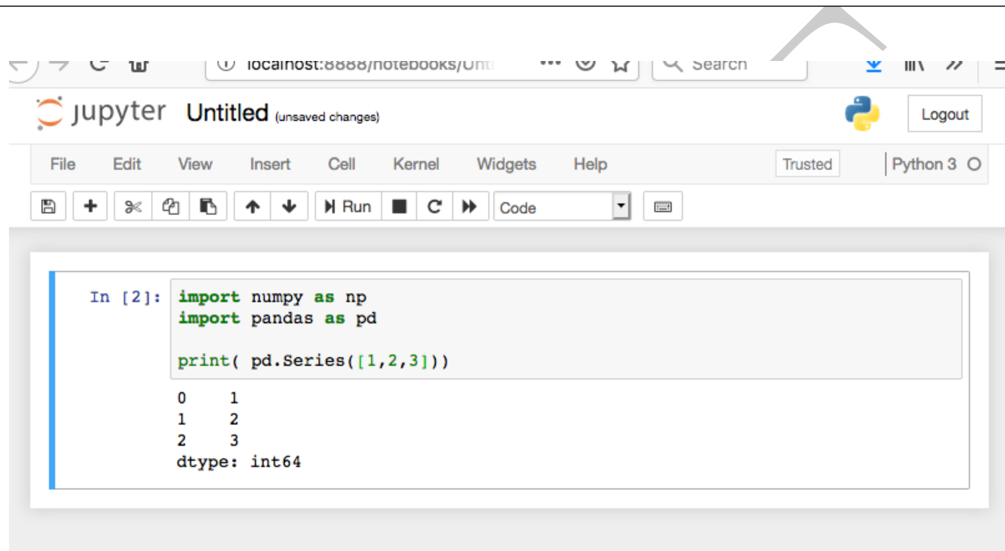
2 HW #0B: Pandas Installation

In order to do the assignments associated with pandas you will need to install Python (and specifically the package pandas) on your computer. The easiest way to do this is by using Anaconda (`https://www.anaconda.com/`) and then use Jupyter Notebooks (`https://jupyter.org/install`).

If you have this installed, you should be able to see a screen like the below and then run the commands below:

```
import numpy as np
import pandas as pd

print( pd.Series([1,2,3]) )
```



DRA

3 HW #0C: MS CAPP Installation instructions

The first assignment is to set up and access the data used in this course.

- Make sure that on Slack channel and on the Canvas page.
- Install PostgreSQL on your computer.[2] Note that installing PostgreSQL can be difficult and I recommend doing some research before jumping in.
  - The data itself can be found in the repo here: `https://github.com/NickRoss/sql-data`.
  - You are welcome to install the PostgreSQL server however you like. The instructions in the repo use docker and set up all the data (including table creation, loading data, etc.). However, if you do not wish to install docker you are welcome to use an alternative method. Two alternatives are: Postgres.app (`https://postgresapp.com/`) and brew (for macs).[3]
  - If you use a non-docker based method you will be required to load the data into the database yourself. Information and specific commands can be found in the data dictionary and the additional instructions at the end of this document.
- You will also be required to install a PostgreSQL client. I personally use one called Postico, but there are many, many others. PopSQL is a fun one to try too, but it requires an internet connection. One important trick when installing is that if you are referring to your local machine the host is "localhost."
- Note that there is nothing to turn in on this assignment.
- **If you installed WITHOUT using docker you need to do the following:**
  - Once PostgresSQL is installed, please create a schema for the stocks database, which can be done using the command below.

    ```
    create schema stocks;
    commit;
    ```

    We will also create a schema for some other datasets that are using in the class, which is "cls" and can be done using the commands below:

    ```
    create schema cls;
    commit;
    ```

  - All of the data required for the homework can be found in the repo and the queries required to load the data onto your Postgres instance can be found in the data dictionary. Broadly speaking to load the data you must:
    1. Have a schema to place the table in (which is what was done in the previous step)
    2. Create a table to load the data into (the CREATE TABLE commands can be found in the data dictionary)
    3. Use a COPY command to move the data from its raw format into the database.
  - Please load the following datasets onto your local SQL instance: (1) stocks.s2010 (2) stocks.s2011, (3) stocks.fnd in order to get access to the stocks data.

DRAFT

[2] I am not IT and am not going to diagnose issues relating to installing your software.

[3] Note that if you are installing with a mac you need to be careful regarding installation instructions for ARM based processors and older models.

4 HW #1A: Basic SQL Querying

The following questions utilize the financial data in the s2010, s2011 and fnd tables. Before beginning the assignment, *please read the data dictionary to better understand the data.* When doing so, keep an eye on data types for different columns as well as table organization.

- If no table information is given, use the 2010 data.
- If the query returns a significant number of rows, please only copy a few rows in your response.
- For those queries which require specifying a date, please use the format 'YYYY-MM-DD' (as in '2010-01-11'), making sure to use single quotes around the date itself.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

Using the 2010 stocks data, write a query that returns the following.

1. All rows and columns relating to AAPL.
2. The date, open and closing price for AAPL on the 7th of January in 2010.
3. Write a query which returns the stock symbol, the date, the open and close price for the top five open prices in 2010 for stocks on the New York Stock Exchange (NYSE).
4. The days when AAPL has a volume more than 20 million and where the high is great than 45 dollars (2010 data)
5. Write a query which returns 3 columns: the return date, stock symbol and volume, but only for stocks that have a volume larger than 200 million in 2010.

Main Problems

1. Write a query which returns all information about about Google (GOOG), NetFlix (NFLX), Amazon (AMZN) and Microsoft (MSFT) in 2010.
2. Consider stocks on the NYSE which had a volume of more than 1 million. Which stocks (symbol and date) had their open price the same as their low and their closing price the same as their high (2010 data). Order them by symbol alphabetically.
3. Consider stocks on the NYSE in 2010 which had a volume of more than 1 million. Which stocks (symbol and date) had their closing price the same as their low and their opening price the same as their high? Sort them by reverse chronological order.
4. Consider stocks on the NYSE in 2010 which had a volume of more than 1 million. Of those days which a stock had either (a) open = low and close = high or (b) open = high and close = low, which symbol and date has the largest volume traded?
5. Which company (ticker symbol) had the highest net income over all the years that are in the FND table?
6. Which company (ticker symbol) had the highest net income in fiscal year 2011 (use the FND table)?

DRAFT

7. Which company (ticker symbol) had the lowest positive net income over all years (use the FND table)?
8. Which company (ticker symbol), which had a net-income per employee over $1,000, had the largest number of employees (over all years)? Keep units in mind (use the FND table)!
9. Which company (ticker symbol) had the lowest, positive, non-zero, net income in fiscal year 2011 (use the FND table)?
10. Of the companies which had more than 1,000 employees in 2011 which had the highest net income per employee in 2011 (use the FND table)?

DRAFT

5 HW #1B: Basic Pandas

Repeat HW #1A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked The following provides a template that you may wish to use.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
import pandas as pd
import numpy as np

df2010 = pd.read_csv( '/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'])

df2011 = pd.read_csv( '/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate',  'opn', 'high', 'low',
        'cls', 'vol', 'exch'])

dffnd = pd.read_csv( '/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])

## Question #1
ans = df2010.loc[(df2010.loc[:, 'symb']=='AAPL'), :]
print(ans.head())

## Question #2
df2010C = df2010.copy()
df2010C = df2010C.loc[(df2010.loc[:, 'retdate'] =='07-Jan-2010') &
        (df2010.loc[:, 'symb']=='AAPL'), :]
df2010C.loc[:, 'diff'] = df2010C.loc[:, 'opn'] - df2010C.loc[:, 'cls']
ans = df2010C
print(ans.head())
```

First Five

Using the 2010 stocks data, write a query that returns the following.

1. All columns relating to AAPL.
2. All columns from the table and a column with the difference between open and close (open - close) for AAPL on the 7th of January.
3. Write a query which returns the stock symbol, the date, the open and close price for the top five differences (open - close) in 2010 for only those stocks on the New York Stock Exchange (NYSE).
4. The days when AAPL has a volume more than 20 million and where the high is $3 or more dollars greater than the low. Write it twice, once to return a series and once as a dataFrame.
5. Write a query which returns 3 columns: the return date, SYMB and volume, but only for stocks that

have a volume larger than 200 million

Main Problems

1. Write a query which returns all information about about Google (GOOG), NetFlix (NFLX), Amazon (AMZN) and Microsoft (MSFT) in 2010.
2. Write a query which returns the date and symbol of the largest "one-day gainer", that is the stock which has the highest close - open on the NYSE.
3. Write a query which returns the date and symbol of the largest "one-day percentage gainer", that is the stock which has the highest (close - open) / open on the NYSE.
4. Consider stocks on the NYSE which had a volume of more than 1 million. Which stocks (symbol and date) had their open price the same as their low and their closing price the same as their high?
5. Consider stocks on the NYSE which had a volume of more than 1 million. Which stocks (symbol and date) had their closing price the same as their low and their opening price the same as their high?
6. Consider stocks on the NYSE which had a volume of more than 1 million. Of those days which a stock had either (a) open = low and close = high or (b) open = high and close = low, which symbol and date has the largest volume traded?
7. Which company (ticker symbol) had the highest net income over all the years that are in the FND table?
8. Which company (ticker symbol) had the highest net income in fiscal year 2011 (use the FND table)?
9. Which company (ticker symbol) had the lowest, non-zero, net income over all years (use the FND table)?
10. Which company (ticker symbol), which had a net-income per employee over $1,000, had the largest number of employees (over all years)? Keep units in mind (use the FND table)!

Even more problems

1. Which company (ticker symbol) had the lowest, non-zero, net income in fiscal year 2011 (use the FND table)?
2. Of the companies which had more than 1,000 employees in 2011 which had the highest net income per employee in 2011 (use the FND table)?

DRAFT

6 HW #2A: Basic Functions

The following questions utilize the financial data in the s2010, s2011 and fnd tables. Before beginning the assignment, *please read the data dictionary to better understand the data.* When doing so, keep an eye on data types for different columns as well as table organization.

- If no table information is given, use the 2010 data.
- If the query returns a significant number of rows, please only copy a few rows in your response.
- For those queries which require specifying a date, please use the format 'YYYY-MM-DD' (as in '2010-01-11'), making sure to use single quotes around the date itself.

In the problems below you may need to use the following definitions:

- **Profit Margin:** Net Income divided by Revenue.
- **Turnover:** Revenue divided by Inventory.
- **Dollar-volume:** This is the dollar value of stocks traded based on the closing price, so equal to the closing price of the shares traded multiplied by the volume.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

DRAFT

First Five

1. Write a query which returns the date and symbol of the largest "one-day gainer" on the NYSE in 2010, that is the stock which has the highest close - open.
2. Return the symbol, return date and the dollar volume traded for the highest dollar volume traded stocks in 2010 on the NYSE.
3. Using the fnd data, which companies (company name), in fiscal year 2010 had a profit margin greater than 20%, turnover more than 2 and more than 10,000 employees?
4. What are the symbols and dollar volume traded for the companies with the top 5 dollar (based on closing price) volume traded on February 3rd 2010 (NYSE only)?
5. Write a query which returns the stock (symbol only) which has the largest (absolute) difference between high and low price for those stocks which have an absolute difference between their high and low of less than $1 dollar and a volume greater than 5,000 (NYSE only in 2010).

Main Problems

1. The "one-day percentage gain" is equal to $\frac{close-open}{open}$. Write a query which returns the date and symbol of the largest one-day percentage gainer of NYSE stocks in 2010.
2. Write a query which returns the date and symbol of the largest one-day percentage gainer for those stocks on the NYSE whose symbol begins with the letter "R" in 2010.
3. Write a query which returns all stocks (symbol and date) with a one-day percentage gain of more than 70 percent whose symbol either begins with R or ends with C.
4. Write a query which returns the stock (symbol) whose second letter (in their symbol) is "T" and is the largest one-day percentage gainer.

5. For those stocks in fiscal year 2010 with a negative net income, which stock (company name) had the largest amount of inventories (fnd table)?
6. Using the fnd table, write a query which returns the company name and the net income for the stock (in 2010) with the largest net income among those stocks with the phrase "data" (case-insensitive) in the company name.
7. Using the fnd table, write a query which returns the top-5 most profitable (highest net income) companies (and their net income) for those companies with either "bank" or "financial" (case-insensitive) in their company name for fiscal year 2010.
8. Using the fnd table, write a query which returns the minimum of ebitda or net income (call it min_profit) and the company name for companies with "apple" (case-insensitive) in their name. Order the results by number of employees from highest to lowest and only include those companies which have all three numeric columns (ebidta, netinc and emp) present.
9. Using the fnd table, write a query which returns squared difference between ebitda and net income (call it sqr_diff) as well as the company name for companies in fiscal year 2010 whose name includes both a Z and a K, but does not contain a C.
10. Write a query which returns the 2 lowest, positive, net incomes (as well as company names) for those companies in fiscal year 2010 with "ING" in their name where the total number of characters in their name is between 5 and 12 (inclusive).

DRAFT

7 HW #3A: Subqueries

Answer the following questions using only the syntax discussed in class. If a year is unspecified, please use the 2010 data and refer to the data dictionary for questions regarding the contents of each table. Be careful when using the fnd data as many of the items in that dataset are scaled by a factor (e.g. in thousands or millions).

Three terms that are defined in this assignment:

- **Profit Margin:** Net Income divided by Revenue.
- **Turnover:** Revenue divided by Inventory.
- **Dollar-volume:** This is the dollar value of stocks traded based on the closing price, so equal to the closing price of the shares traded multiplied by the volume.

For each question, please provide the query which will generate the result.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

1. Using the daily stock data from 2010, return a list of the unique trading days in 2010.
2. Using the 2010 data return the stock (symbol), the date and the dollar-volume for the stock with the largest dollar-volume traded on the NYSE (on a single day).
3. Using the 2010 data, return the stock (symbol only) with the largest volume on Jan 11th that also appears on Dec 1st.
4. Using the 2010 data, return the stock symbol and a column called "HFlag" which is equal to 1 if the high - low is greater than 1 and zero otherwise. Only return those companies whose stock symbol begins or ends with "A".
5. Write a query which returns (a) the date, (b) closing price and (c) a flag ("gt30") which is equal to "1" when the closing price is greater than $30.00 and "0" otherwise for "AAPL" in 2010.

Main Problems

1. Return the list of symbols that exist in 2011, but not 2010.[4]
2. Using the fnd data, return company name, year and the a column called "HFlag" which is equal to 1 if the company has a net income larger than $1 Billion dollars and 0 otherwise. Only include those companies whose name begins with "B".
3. Using the fnd data, which ticker symbols have a net income to employee ratio greater than $1,000 in fiscal year 2010 and also have a net income between 20 and 30 million dollars in 2011?
4. The lowest five symbols by volume from Janaury 11th, 2010 that have a volume between 1 million and 10 million on December 1st, 2011. In other words, of those stocks which had between 1 and 10 million shares traded on December 1st, 2011, which five have the lowest volume traded on January 11th, 2010.

[4] If this is slow, try using distinct and see what happens. Any ideas why this may happen?

DRAFT

5. Of the stocks (symbols) that existed in 2011, but not in 2010, which had the highest closing price in 2011?
6. Which symbols were in the top 500 of dollar volume on the 2nd, 3rd and 4th days of February 2011 (The stock needs to be in the top 500 for all days)?
7. Of the symbols that had volume between 100,000 and 1,000,000 on the 2nd and 3rd of February 2011, which had volume greater than 5,000,000 on the 4th on February?
8. Write a query to generate the following dataset:
   - company name, ticker symbol, revenue for all companies whose name begin with "A" or "a"
   - A column, revflag which is 1 if revenue is greater than $25,000,000 and 0 otherwise.
9. Write a query to generate the following dataset:
   - company name, ticker symbol, revenue, inventory and employee information from fiscal year 2010
   - A column called turnflag which is 1 for companies with turnover greater than 2, 0 otherwise
   - For a company to be included it must have revenue, inventory and employee all greater than zero for both 2010 and 2011

DRAFT

Additional Problems

1. Of the stocks (ticker symbols) that have a net income to revenue ratio (called a profit margin) greater than 20%, which have more than 25,000 employees in fiscal year 2011?
2. We define revenue divided by inventory as the turnover. It expresses how many times the inventory has turned-over during the year in the form of sales. For companies (ticker symbols) with revenue between 1 and 2 million dollars in 2010, what company has the highest turnover in 2011?
3. Of the stocks (ticker symbol) that have profit margin greater than 20% in 2010, which had a profit margin greater than 30% in fiscal year 2011?
4. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and more than 1,000 employees in 2011, what is the highest profit margin in fiscal year 2011 and what is the ticker symbol?
5. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and more than 1,000 employees in 2011, what is the lowest profit margin in fiscal year 2011?
6. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and between 1,000 and 2,000 employees in 2011, what is the highest profit margin in fiscal year 2011 and what is the ticker symbol?
7. Of the companies (ticker symbols) with turnover between 1 and 2 in 2010, which companies also had a net income to employee ratio greater than $1,000 in 2010?
8. Of the companies (ticker symbols) with turnover between 1 and 2 in 2010, which companies also had a net income to employee ratio greater than $1,000 in 2011?
9. Write a select statement to generate the following dataset:
   - company name, ticker symbol, revenue, inventory and employee information from both 2010 and 2011 fiscal years.

- A column called invtflag which is equal to 1 for companies with turnover between 2 and 3, 2 for turnover between 3 and 4 and 5 for turnover greater than 4 and zero otherwise.
- A column called invtProfit which is equal to 1 for companies with less than 20% profit margin and turnover greater than 2, 2 for companies with profit margin greater than 40% and turnover greater than 2 and 0 otherwise.
- A column called EmployeeProfit which is equal to 0 for companies that have profit margins between 20% and 40% and have more than 10,000 employees, is equal to a company's profit margin if the margin is less than 20%, is equal to twice the number of employees (if it exists) if the profit margin is greater than 40% and is -1 otherwise.

DRAFT

8 HW #3B: Subqueries in Pandas

Repeat HW #3A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked. The same requirements as in HW #1B apply.

Three terms that are defined in this assignment:

- **Profit Margin:** Net Income divided by Revenue.
- **Turnover:** Revenue divided by Inventory.
- **Dollar-volume:** This is the dollar value of stocks traded based on the closing price, so equal to the closing price of the shares traded multiplied by the volume.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
## Initial Information
import pandas as pd
import numpy as np
df2010 = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls' , 'vol', 'exch'])

df2011 = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate',  'opn', 'high', 'low',
        'cls', 'vol', 'exch'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])
```

DRAFT

First Five

1. Using the daily stock data from 2010, return an array of the unique trading days in 2010.
2. Return the symbol, return date and the dollar volume traded for the highest dollar volume traded stocks in 2010 on the NYSE.
3. Using the 2010 data, return the stock (symbol only) with the largest volume on Jan 11th that also appears on Dec 1st.
4. Using the 2010 data, return the stock symbol and a column called “HFlag” which is equal to 1 if the high - low is greater than 1 and zero otherwise. Only return those companies whose stock symbol begins or ends with “A”.
5. Write a query which returns (a) the date, (b) closing price and (c) a flag (“gt30”) which is equal to “1” when the closing price is greater than $30.00 and “0” otherwise for “AAPL” in 2010.

Main Problems

1. Return the list of symbols that exist in 2011, but not 2010.

2. Using the fnd data, return company name, year and the a column called "HFlag" which is equal to 1 if the company has a net income larger than $1 Billion dollars and 0 otherwise. Only include those companies whose name begins with "B".
3. Using the fnd data, which ticker symbols have a net income to employee ratio greater than $1,000 in fiscal year 2010 and also have a net income between 20 and 30 million dollars in 2011?
4. Using the fnd data, which companies (company name), in fiscal year 2010 had a profit margin greater than 20%, turnover more than 2 and more than 10,000 employees?
5. The lowest five symbols by volume from Janaury 11th, 2010 that have a volume between 1 million and 10 million on December 1st, 2011. In other words, of those stocks which had between 1 and 10 million shares traded on December 1st, 2011, which five have the lowest volume traded on January 11th, 2010.
6. Of the stocks (symbols) that existed in 2011, but not in 2010, which had the highest closing price in 2011?
7. Which symbols were in the top 500 of dollar volume on the 2nd, 3rd and 4th days of February 2011 (The stock needs to be in the top 500 for all days)?
8. Of the symbols that had volume between 100,000 and 1,000,000 on the 2nd and 3rd of February 2011, which had volume greater than 5,000,000 on the 4th on February?
9. Generate the following dataset:
   - company name, ticker symbol, revenue for all companies whose name begin with "A" or "a"
   - A column, revflag which is 1 if revenue is greater than $25,000,000 and 0 otherwise.
10. Generate the following dataset:
    - company name, ticker symbol, revenue, inventory and employee information from fiscal year 2010
    - A column called turnflag which is 1 for companies with turnover greater than 2, 0 otherwise
    - For a company to be included it must have revenue, inventory and employee all greater than zero for both 2010 and 2011

DRAFT

Additional Problems

1. Of the stocks (ticker symbols) that have a net income to revenue ratio (called a profit margin) greater than 20%, which have more than 25,000 employees in fiscal year 2011?
2. We define revenue divided by inventory as the turnover. It expresses how many times the inventory has turned-over during the year in the form of sales. For companies (ticker symbols) with revenue between 1 and 2 million dollars in 2010, what company has the highest turnover in 2011?
3. Of the stocks (ticker symbol) that have profit margin greater than 20% in 2010, which had a profit margin greater than 30% in fiscal year 2011?
4. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and more than 1,000 employees in 2011, what is the highest profit margin in fiscal year 2011 and what is the ticker symbol?
5. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and more than 1,000 employees in 2011, what is the lowest profit margin in fiscal year 2011?

6. Of the stocks (ticker symbols) that have a net-income to employee ratio greater than $1,000 in fiscal year 2010 and between 1,000 and 2,000 employees in 2011, what is the highest profit margin in fiscal year 2011 and what is the ticker symbol?
7. Of the companies (ticker symbols) with turnover between 1 and 2 in 2010, which companies also had a net income to employee ratio greater than $1,000 in 2010?
8. Of the companies (ticker symbols) with turnover between 1 and 2 in 2010, which companies also had a net income to employee ratio greater than $1,000 in 2011?
9. Write a select statement to generate the following datasets:
   - company name, ticker symbol, revenue, inventory and employee information from both 2010 and 2011 fiscal years.
   - A column called invtflag which is equal to 1 for companies with turnover between 2 and 3, 2 for turnover between 3 and 4 and 5 for turnover greater than 4 and zero otherwise.
   - A column called invtProfit which is equal to 1 for companies with less than 20% profit margin and turnover greater than 2, 2 for companies with profit margin greater than 40% and turnover greater than 2 and 0 otherwise.
   - A column called EmployeeProfit which is equal to 0 for companies that have profit margins between 20% and 40% and have more than 10,000 employees, is equal to a company's profit margin if the margin is less than 20%, is equal to twice the number of employees (if it exists) if the profit margin is greater than 40% and is -1 otherwise.
10. What are the symbols and dollar volume traded for the companies with the top 5 dollar volume traded (based on closing price) on February 3rd 2010 (NYSE only)?

DRAFT

9 HW #4A: Aggregation

Answer the following questions using only the syntax discussed in class. If a year is unspecified, please use the 2010 data and refer to the data dictionary for questions regarding the contents of the data.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

1. What is the total number of rows in the 2010 database?
2. How many unique symbols are there in 2010?
3. What is the minimum closing price for a stock when it had a volume greater than 1,000,000 shares in 2010?
4. Return the average closing price for all stocks from the NYSE in 2010?
5. Return the average closing price for all stocks and the total number of rows by the exchange of each stock for 2010. Order the results from lowest to highest average closing price. This should return two rows and three columns (exchange, average closing price and total number of rows)

Main Problems

1. Which symbols have less than 50 rows in 2010?
2. How many symbols have less than 50 rows in 2010?
3. Write a query which returns one row and two columns. The first column should contain the number of symbols which have less than 50 rows in 2010 and the second column should have the number of symbols with more than 100 rows in 2010.
4. Write a query which returns two column and two rows. The first column should be named "numtype" which should be equal to "less than 50" or "more than 100" and the second column should have the number of unique symbols which correspond to this condition. In other words, the same numbers as the previous problem, transposed with an column providing a description.
5. Write a query which returns three rows and two columns. The first column should contain the average yearly total traded volume for symbols which had (1) more than 100 trading days (2) less than 50 trading days and (3) between 50 and 100 trading days. The other column should identify each row and be called "numType."
6. Write a query which returns three rows and two columns. The first column should contain the average *daily* traded volume for symbols which had (1) more than 100 trading days (2) less than 50 trading days and (3) between 50 and 100 trading days. The other column should identify each row and be called "numType."
7. How many of the symbols had a day where the dollar volume (closing price multiplied by number of shares traded) was greater than 100 million dollars in 2010?
8. What percentage of the symbols had a day where the dollar volume of shares traded was greater than 100 million dollars in 2010?

DRAFT

9. Using only the SUM, AVG and COUNT aggregate functions, compute the covariance between the the closing price and volume in 2010.

DRAFT

10 HW #4B: Aggregation in Pandas

Repeat HW #4A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked.

Answer the following questions using only the syntax discussed in class. If a year is unspecified, please use the 2010 data and refer to the data dictionary for questions regarding the contents of the data.

Here are the statements that will load the data, note that you will need to change the directory.

```
## Initial Information
import pandas as pd
import numpy as np
df2010 = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls' , 'vol', 'exch'])

df2011 = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate',  'opn', 'high', 'low',
        'cls', 'vol', 'exch'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])
```

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

DRAFT

First Five

1. What is the total number of rows in the 2010 database? Return this as an integer.
2. How many unique symbols are there in 2010? Return this as an integer.
3. What is the minimum closing price for a stock when it had a volume greater than 1,000,000 shares in 2010?
4. Return the average closing price for all stocks from the NYSE in 2010?
5. Return the average closing price for all stocks and the total number of rows by the exchange of each stock for 2010. Order the results from lowest to highest average closing price. This should return two rows and three index/value columns (exchange, average closing price and total number of rows).

Main Problems

1. Which symbols have less than 50 rows in 2010?
2. How many symbols have less than 50 rows in 2010?
3. Write a query which returns one row and two columns (DataFrame or Series). The first column should contain the number of symbols which have less than 50 rows in 2010 and the second column should have the number of symbols with more than 100 rows in 2010.

4. Write a query which returns two column and two rows (either series or a DataFrame). The first column should be equal to “lessThan50” or “moreThan100” and the second column should have the number of unique symbols which correspond to this condition. In other words, the same numbers as the previous problem, transposed with an column providing a description.
5. Write a query which returns three rows and two columns (note that one column maybe an index). One column should contain the average yearly total traded volume for symbols which had (1) more than 100 trading days (2) less than 50 trading days and (3) between 50 and 100 trading days. The other column should identify each row and be called “numType.”
6. Write a query which returns three rows and two columns. The first column should contain the average *daily* traded volume for symbols which had (1) more than 100 trading days (2) less than 50 trading days and (3) between 50 and 100 trading days. The other column should identify each row and be called “numType.”
7. How many of the symbols had a day where the dollar volume (closing price multiplied by number of shares traded) was greater than 100 million dollars in 2010?
8. What percentage of the symbols had a day where the dollar volume of shares traded was greater than 100 million dollars in 2010?

DRAFT

11 HW #5A: Aggregate Functions and Dates

Answer the following questions using only the syntax discussed in class. If a year is unspecified, please use the 2010 data and refer to the data dictionary for questions regarding the contents of the data.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

The queries below rely on information from the 2010 stock return data.

1. Which day of the week (0,1,2,...) had the largest number of shares traded?
2. Which day of the week (0,1,2,..) has the highest average shares traded?
3. Which day of the week-month (January-Monday, January-Tuesday, etc.) combination had the highest average return (close - open)? Note that both day of the week and month can be kept as integers.
4. Write a query which returns 3 columns and 5 rows with each row should represent a day of the week. One column should be the English day of the week ("Monday,", "Tuesday," etc.) while the next column should be equal to the average number of shares traded on that day from stocks that have a volume traded between 1 million and 2 million shares on that day ("C2"). The final column ("C3") should be the average number of shares traded on that day from stocks that had a volume traded outside of 1 million to 2 million.
5. Write a query which returns the maximum closing price for each symbol in 2010, sorting the results from from high-to-low closing price.

Main Questions

1. Which quarter in 2010 has the most trading days?[5]
2. Write a query which returns symbol and a column "DFlag", which is equal to 1 if the max closing price in 2010 is larger than 100, 2 if the max closing price in 2010 is between 50 and 100 and 3 if the max closing price is less than 50. There should be one row per symbol.
3. Write a query which returns the number of distinct symbols of each type of Dflag (from the previous problem). This should be 3 rows and 2 columns (one of the columns should indicate what each row means).
4. Write a query which returns the number of distinct symbols of each type of Dflag (from the previous problems), this should be 3 columns and a single row.
5. Calculate the number of distinct trading days per month in 2010. This should return 12 rows with 2 columns.
6. For each symbol, calculate the difference between the maximum and minimum closing price for December, 2010. Only include those stocks with 22 observations (there are 22 trading days in December, 2010).

[5]Define Q1 as Jan-Mar, Q2 as Apr-Jun, etc.

DRAFT

7. Calculate the average difference between the maximum and minimum closing price for Tuesdays in January, 2010 for stocks on NYSE. The max and min should be calculated per-stock and then averaged. Only include those stocks with 4 observations which fulfill the criteria.[6]
8. Calculate the average closing price for Tuesday in January 2010 from the NYSE. Only include those stocks with 4 observations which fulfill the criteria. In other words, calculate the average price for each stock and then take the average of that number.
9. Calculate the average closing price for all stocks on the NYSE, by month, in 2010. Only include those stocks which have a closing price greater than $100 in *2011*.
10. Calculate the average closing price in 2010 for all stocks (NYSE only) which are "not extreme". We define a stock as not extreme if the closing price is less than .1% of the max closing price (for all stocks) for the entire year. In other words, identify those stocks which are not extreme and then calculate their average price.

DRAFT

[6] There are 4 Tuesday trading days in January, 2010.

12 HW #5B: Aggregate Functions and Dates

Repeat HW #5A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked.

The queries below rely on information from the stock return data. To load the data use the following commands. **Note: these are different than the previous commands because they load retdate as a date, rather than a string**

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
## Initial Information
import pandas as pd
import numpy as np

df2010D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

df2011D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])
```

Draft

First Five

1. Which day of the week (0,1,2,...) had the largest number of shares traded?
2. Which day of the week (0,1,2,..) has the highest average shares traded?
3. Which day of the week-month (January-Monday, January-Tuesday, etc.) combination had the highest average return (close - open)? Note that both day of the week and month can be kept as integers.
4. Write a query which returns 3 columns and 5 rows with each row should represent a day of the week. One column should be the english day of the week ("Monday,", "Tuesday," etc.) while the next column should be equal to the average number of shares traded on that day from stocks that have a volume traded between 1 million and 2 million shares on that day ("C2"). The final column ("C3") should be the average number of shares traded on that day from stocks that had a volume traded outside of 1 million to 2 million.
5. Write a query which returns the maximum closing price for each symbol in 2010, sorting the results the final table from from high-to-low closing price.

Main Questions

1. Which quarter in 2010 has the most trading days?[7]

[7] Define Q1 as Jan-Mar, Q2 as Apr-Jun, etc.

2. Write a query which returns symbol and a column "DFlag", which is equal to 1 if the max closing price in 2010 is larger than 100, 2 if the max closing price in 2010 is between 50 and 100 and 3 if the max closing price is less than 50. There should be one row per symbol.
3. Write a query which returns the number of distinct symbols of each type of Dflag (from the previous problem). This should be 3 rows and 2 columns (one of the columns should indicate what each row means).
4. Write a query which returns the number of distinct symbols of each type of Dflag (from the previous problems), this should be 3 columns and a single row.
5. Calculate the number of distinct trading days per month in 2010. This should return 12 rows with 2 columns.
6. For each symbol, calculate the difference between the maximum and minimum closing price for December, 2010. Only include those stocks with 22 observations (there are 22 trading days in December, 2010).
7. Calculate the average difference between the maximum and minimum closing price for Tuesdays in January, 2010 for stocks on NYSE. The max and min should be calculated per-stock and then averaged. Only include those stocks with 4 observations which fulfill the criteria.[8]
8. Calculate the average closing price for Tuesday in January 2010 from the NYSE. Only include those stocks with 4 observations which fulfill the criteria. In other words, calculate the average price for each stock and then take the average of that number.
9. Calculate the average closing price for all stocks on the NYSE, by month, in 2010. Only include those stocks which have a closing price greater than $100 in *2011*.
10. Calculate the average closing price in 2010 for all stocks (NYSE only) which are "not extreme". We define a stock as not extreme if the closing price is less than .1% of the max closing price (for all stocks) for the entire year. In other words, identify those stocks which are not extreme and then calculate their average price.

DRAFT

[8] There are 4 Tuesday trading days in January, 2010.

13 HW #6A: SQL Joins (I)

The following questions utilize the financial data in the s2010, s2011 and fnd tables. Before beginning the assignment, *please read the data dictionary to better understand the data.* When doing so, keep an eye on data types for different columns as well as table organization.

- If no table information is given, use the 2010 data.
- If the query returns a significant number of rows, please only copy a few rows in your response.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

1. Using a JOIN, create a dataset which contains symbol, the max closing price for that symbol from 2010 and the max closing price for that symbol from 2011. This should only include those symbols which are in both 2010 and 2011. Are you sure that both sides are unique? Why?
2. Using a LEFT JOIN, create a dataset which contains the following information: symbol, the last day it is traded in 2011 and the last day it is traded in 2010. Make sure to include all rows from 2011 and only those matching from 2010. There should be one row per symbol.
3. Using a cross join, create a dataset which contains every possible combination of symbol (in 2010) and return date (in 2010).
4. Write a query which returns the number of rows in the above query. How does this compare to the number of rows in the 2010 dataset? Does this make sense?
5. Write a query which has 12 rows and 3 columns. The first column should be Month (1,2,3...,12) the second column should be the number of rows from that month in 2010 and the third column should be the number of rows from that month in 2011.

Main Problems

1. Using a LEFT JOIN, count the number of symbols which are in 2010, but not in 2011.
2. For each symbol, return the closing price on the first day that it is traded in 2010.
3. For each symbol, return the closing price on both the first day and last day that it is traded in 2010.
4. Create a dataset which contains 4 columns: the symbol, the retdate, the closing price and the closing price on the day after. Note that this dataset should *only* include Monday to Tuesday transitions, so retdate there should only be one row per-symbol per-Monday in the dataset. Specifically, if there are 50 trading weeks in a year and assuming that a symbol is traded every day, there would be 50 observations for that symbol
5. By matching the fnd data and the stocks 2010 data create a table which contains three columns and one row. The columns should represent the number of *unique* symbols which (a) are in both datasets, (b) are only in the 2010 dataset and (c) are only in the fnd data. Make sure to ignore all observations which are missing ticker symbols.

DRAFT

6. By combining the fnd and the stocks 2010 data, generate a dataset which contains the number of unique symbols of each of the three types in the previous problem. This time return two columns and three rows (one of the columns should describe what data is in the row).
7. Create a dataset which is 5 rows by 3 columns. The first column should be DOW, the second column should be the average closing price of all stocks from 2010 on that day of the week and the third should be the average price of all stocks from 2011 for that day of the week.
8. We want to divide all stocks by the following criteria: if their max closing price in 2010 was less than 50, between 50 and 100 (inclusive) and more than 100. Return a table which contains the average net income (from fyear 2010) for each type of stock. Note that net income can be found in the fnd table and, if there are two net-income values for a particular ticker symbol, take the max. Only include those symbols in both datasets (fnd and s2010) that do not have a missing net income.

Extra Problems

1. Create a dataset which contains the first day that each symbol is traded in 2010, the last day that the symbol is traded in 2011 and only includes those symbols which are in both 2010 and 2011.
2. For those symbols which had a closing price larger than $100 *anytime* in 2010, return the symbol, first day that it was traded in 2010 and all the dates that it had a closing price larger than $200 in 2010. If the symbol was never above $200, return no rows for it.
3. What are the first and last date listed for each symbol in 2010? Be careful to return this for *each* symb.
4. For each symbol that appears anywhere in 2010, calculate the number of missing trading days that it has in each month in 2010. This should return three columns: symbol, month, number of missing values.
5. Create a dataset which is 10 rows by 3 columns. The first column should be the year, the second column should be the day-of-the-week and the third column should be the average closing price of all stocks for that day-of-the-week. Include both 2010 and 2011.
6. How many cars (total), on an average *day*, go through each toll plaza in both directions combined (return a row for each toll plaza)? Make sure to sum up to the *day* level before computing the average.
7. Which day-of-the-week (Monday, Tuesday, etc.) has the highest number of cars going through Plaza #1, both directions combined, with EZ pass? This should be the total number of cars over the entire time period in the dataset.
8. Which day-of-the-week-plaza combination has the lowest percentage of users cars using the EZ pass in the outbound direction? In other words, if you look at outbound cars through each plaza, which day of the week has the lowest percentage of cars using EZ pass. You can compute the percentage over the entire time period.
9. Calculate the average number of cars going through Plaza #1, outbound, with EZ pass for each day-of-the-week. This should be a *daily* average and should return 7 rows.
10. In an average week on Plaza #1 with EZ pass (outbound), what percentage of cars go through each day? (E.g. basically the above, but this time percent of total).
11. For each plaza, what was the change (percent) in average number of cars on a Monday using EZ-pass in both directions, between 2015 and 2016? (Calculate the average number of cars for a Monday in 2015 and 2016 and then calculate the percentage change based off of that.)

DRAFT

12. Calculate, for each hour, plaza and day-of-the-week (so $7 \cdot 24$ rows per plaza), the ratio of inbound to outbound traffic.
13. Using a join, create a dataset with three columns and 7 rows. The first column should be the DOW, the second column should be the average number of cars, per-day-of-the-week, through toll Plaza #1 in either direction with an EZ pass in 2016 and the final column should be the average number of cars, per-day-of-the-week, through tool Plaza #2 with an EZ pass in 2015.
14. Create a dataset which contains twenty-four rows and two columns. The first column represents the hour and the second column represents the max number of EZ pass cars, during that hour, outbound, through Plaza #1.
15. Create a dataset which contains 24x7 rows and two columns. The first column represents the DOW-hour combination (you may need to combine two columns using "||" or the concatenate operator) and the second represents the max number of EZ pass cars, during that hour-day, through Plaza #1 in the outbound direction.
16. Using at least one join, create a dataset which contains twenty-four rows and 4 columns. Each row should represent an hour, and the first column should be an hour identifier. Column #2 should contain the maximum number of EZ pass cars, in the inbound direction, through Plaza #1 during that hour, Column #3 should contain the minimum number of outbound EZ-pass cars, during that hour, through Plaza #2 and Column #3 should be the maximum number of EZ-pass cars in either directions combined, during that hour, on Plaza 3.
17. Create a dataset which contains the following columns: hour, day-of-the-week, plaza, the ratio of inbound to outbound traffic in 2014 and the ratio of inbound to outbound traffic in 2013.
18. For the day with the most traffic (inbound, outbound and both payment types combined), calculate the ratio of inbound to outbound traffic over the entire dataset (not by plaza), for each hour. Return three columns, the day-of-the-week of that date, hour and the percent for that hour.

DRAFT

14 HW #6B: Pandas Joins (I)

Repeat HW #6A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked.

The queries below rely on information from the stock return data. To load the data use the following commands. **Note: these load retdate as a date, rather than a string**

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
## Initial Information
import pandas as pd
import numpy as np

df2010D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

df2011D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])

dfMTA = pd.read_csv('../sql-data/raw_data/mta/MTA_Hourly.tdf', sep='\t',
        engine='python', names=['plaza', 'mtadt', 'hr', 'direction', 'vehiclesez',
        'vehiclescash'])

dfTrans = pd.read_csv('../sql-data/raw_data/soapData.tdf', sep='\t',
        engine='python', names = ['orderid', 'userid', 'trans', 'type', 'local',
        'trans_dt', 'units', 'coupon', 'months', 'amt' ])
```

First Five

1. Using a join, create a dataset which contains symbol, the max closing price for that symbol from 2010 and the max closing price for that symbol from 2011. This should only include those symbol which are in both 2010 and 2011. Are you sure that both sides are unique? Why?
2. Using a LEFT JOIN, create a dataset which contains the following information: symbol, the last day it is traded in 2011 and the last day it is traded in 2010. Make sure to include all rows from 2011 and only those matching from 2010. There should be one row per symbol.
3. Using a cross join, create a dataset which contains every possible combination of symbol (in 2010) and return date (in 2010).
4. Write a query which returns the number of rows in the above query. How does this compare to the number of rows in the 2010 dataset? Does this make sense?
5. Write a query which has 12 rows and 3 columns. The first column should be Month (1,2,3...,12) the

second column should be the number of rows from that month in 2010 and the third column should be the number of rows from that month in 2011.

Main Problems

1. Using a LEFT JOIN, count the number of symbols which are in 2010, but not in 2011.
2. For each symbol, return the closing price on the first day that it is traded in 2010.
3. For each symbol, return the closing price on both the first day and last day that it is traded in 2010.
4. Create a dataset which contains 4 columns: the symbol, the retdate, the closing price and the closing price on the day after. Note that this dataset should *only* include Monday to Tuesday transitions, so retdate there should only be one row per-symbol per-Monday in the dataset. Specifically, if there are 50 trading weeks in a year and assuming that a symbol is traded every day, there would be 50 observations for that symbol
5. By matching the fnd data and the stocks 2010 data create a table which contains three columns and one row. The columns should represent the number of *unique* symbols which (a) are in both datasets, (b) are only in the 2010 dataset and (c) are only in the fnd data. Make sure to ignore all observations which are missing ticker symbols.
6. By combining the fnd and the stocks 2010 data, generate a dataset which contains the number of unique symbols of each of the three types in the previous problem. This time return two column and three rows (one of the columns should describe what data is in the row).
7. Create a dataset which is 5 rows by 3 columns. The first column should be DOW, the second column should be the average closing price of all stocks from 2010 on that day of the week and the third should be the average price of all stocks from 2011 for that day of the week.
8. We want to divide all stocks by the following criteria: if their max closing price in 2010 was less than 50, between 50 and 100 (inclusive) and more than 100. Return a table which contains the average net income (from fyear 2010) for each type of stock. Note that net income can be found in the fnd table and, if there are two net-income values for a particular ticker symbol, take the max. Only include those symbols in both datasets (fnd and s2010) and that do not have a missing net income.

Extra Problems

1. Create a dataset which contains the first day that each symbol is traded in 2010, the last day that the symbol is traded in 2011 and only includes those symbols which are in both 2010 and 2011.
2. For those symbols which had a closing price larger than \$100 *anytime* in 2010, return the symbol, the first day that it was traded in 2010 and all the dates that it had a closing price larger than \$200 in 2010. If the symbol was never above \$200, return no rows for it.
3. What are the first and last date listed for each symbol in 2010? Be careful to return this for *each* symb.
4. For each symbol that appears anywhere in 2010, calculate the number of missing trading days that it has in each month in 2010. This should return three columns: symbol, month, number of missing values.
5. Create a dataset which is 10 rows by 3 columns. The first column should be the year, the second column should be the day-of-the-week and the third column should be the average closing price of all stocks for that day-of-the-week. Include both 2010 and 2011.

DRAFT

15 HW #7A: SQL Joins (II)

Please answer the following questions, making sure to only use the syntax from class.[9]

Before beginning the assignment make sure that you have indexes applied to the symbol and return date variables in both stock tables (otherwise the queries will take an eternity). The following commands will create the indexes necessary to complete the assignment.

```
create index s2010_symb_retdate on stocks.s2010 (symb, retdate);
create index s2011_symb_retdate on stocks.s2011 (symb, retdate);
```

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

First Five

1. Write a query which returns the symbol, date, volume and the total volume traded for that stock up to (but not including) that *day's* volume. Only consider 2010 data. Make sure that if there is no previous volume that the cumulative volume is set to *zero* and isn't null.
2. Write a query which returns the number of days that the stock has been traded, cumulatively, in 2010. Specifically, the first time that the stock appears it should be "1", the second date that it exists it should be "2". This should return a table with 3 columns (symb, retdate and cumulative trading days) and should have the same number of rows as the original s2010 dataset.
3. For each exchange in 2010 return the stock with the highest total traded volume in the year 2010. This should return two rows (one for each exchange) and 3 columns (exchange name, symbol and total volume traded for that year)
4. For each exchange in 2010 return the stock with the *second* highest total traded volume in the year 2010. This should return two rows (one for each exchange) and 3 columns (exchange name, symbol and total volume traded for that year)
5. Create a dataset which contains the following columns: symbol, date that the symbol first appears in 2010 and the total volume traded in the first 35 days that the stock is traded in 2010. If a stock has a first date late in the year, ignore the spill over into 2011.

Main Problems

1. Using the data from 2010, write a query which returns the seven day moving average for each stock's closing price. Only look at stocks whose symbols begin with the letter "A". Note that this should *not* be the last seven points, but instead the last seven *days*, not including the current day.
2. For each stock from 2010, write a query which returns the symbol, the closing price, the return date and the closing price on the previous day it was traded. Note that you just want to take the price from the previous row, if the rows are ordered by return date. Also, only do this for stocks that begin with the letter 'A'.
3. For each symbol in 2010, return the day(s) where it has its highest volume traded[10]

[9] Specifically if you decided to look ahead, analytic functions are not to be used to answer these questions.
[10] There could be multiple days for a symb.

DRAFT

4. Using only a single join, for each symbol, return the closing price on the first and last day that the stock is traded in 2010.
5. How many missing days are there in total? Make sure to only count missing days **after** a symbol has been in the data. So if a stock doesn't appear in the data until February, January does not count as missing. If a stock leaves the market before the end of the year, you can either count the days past their exit as missing or as not missing, just be consistent across all stocks.[11]
6. For each symbol that appears in 2011, calculate the number of missing trading days that it has in January 2010.
7. Write a query which returns the userid, trans_dt, amt, and the total amount the user has spent up to (but not including) that *day's* transaction. Make sure that if there is no previous transaction the amount is set to *zero* and isn't null.
8. Write a query which returns a purchase number for each order. In other words, for each row return the amount, userid, date and the number of the sale, incrementing from one for each order.
9. For each local in the table, return the most common month of an order.
10. For each local in the table, return the *second* most common month of an order.
11. Create a dataset which has the following information: (1) userid, (2) date of first transaction and (3) number of transactions within the first 35 days of their first transaction.

DRAFT

[11] In other words, if a stock leaves the data it maybe because the stock delisted, in which case the data is not missing.

16 HW #7B: Pandas Joins (II) [TBD]

NEEDS TO BE REWRITTEN AK

Repeat HW #7A, this time using Pandas. In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked.

The queries below rely on information from the stock return data. To load the data use the following commands. **Note: these load retdate as a date, rather than a string**

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
## Initial Information
import pandas as pd
import numpy as np

df2010D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

df2011D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])

dfMTA = pd.read_csv('../sql-data/raw_data/mta/MTA_Hourly.tdf', sep='\t',
        engine='python', names=['plaza', 'mtadt', 'hr', 'direction', 'vehiclesez',
        'vehiclescash'])

dfTrans = pd.read_csv('../sql-data/raw_data/soapData.tdf', sep='\t',
        engine='python', names = ['orderid', 'userid', 'trans', 'type', 'local',
        'trans_dt', 'units', 'coupon', 'months', 'amt' ])
```

First Five

1. Write a query which returns the userid, trans_dt, amt and the total amount that the user has spent up to (but not including) that *day's* transaction. Make sure that if there is no previous transaction that the amount is set to *zero* and isn't null.
2. Write a query which returns a purchase number for each order. In other words, for each row return the amount, userid, date and the number of the sale, incrementing from one for each order.
3. For each local in the table, return the most common month of an order.
4. For each local in the table, return the *second* most common month of an order.
5. Create a dataset which has the following information: (1) userid, (2) date of first transaction and (3) number of transactions within the first 35 days of their first transaction.

Main Problems

1. Using the data from 2010, write a query which returns the seven day moving average for each stock's closing price. Only look at stocks whose symbols begin with the letter "A". Note that this should *not* be the last seven points, but instead the last seven *days*, not including the current day.
2. For each stock from 2010, write a query which returns the symbol, the closing price, the return date and the closing price on the previous day it was traded. Note that you just want to take the price from the previous row, if the rows are ordered by return date. Also, only do this for stocks that begin with the letter 'A'.
3. For each symbol in 2010, return the day(s) where it has its highest volume traded[12]
4. Return the closing price on the first and last day that the stock is traded in 2010.
5. How many missing days are there total? Make sure to only count missing days **after** a symbol has been in the data. So if a stock doesn't appear in the data until February, January does not count as missing. If a stock leaves the market before the end of the year, you can either count the days past their exit as missing or as not missing, just be consistent across all stocks.[13]
6. For each symbol that appears in 2011, calculate the number of missing trading days that it has in January 2010.

DRAFT

[12] There could be multiple days for a symb.

[13] In other words, if a stock leaves the data it maybe because the stock delisted, in which case the data is not missing.

17 HW #8AO: SQL Window Functions: [TBD]

THIS ONE IS REPLACED AND NEEDS TO BE FIXED

Using only the functions and syntax that we have learned in class, please provide a query to answer the following questions. If a dataset is not specified, please use the 2010 dataset. **Do not create any tables or views.**

Before beginning the assignment, *please read the data dictionary to better understand the data.* When doing so, keep an eye on data types for different columns as well as table organization.

- If no year information is provided for a financial question, assume 2010.
- If the query returns a significant number of rows, please only copy a few rows in your response.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

DRAFT

First Five

1. What is the median daily closing price on January 4th, 2010?
2. For stocks in 2010, write a query which creates a dataset containing closing price, symbol, retdate and the nominal change between yesterday's closing price and today's opening price. Ignore holes in the data, so that if the stock misses a day the change in price is from the last time listed.
3. Write a query which returns five columns: symbol, return date, closing price, the moving average of the price (covering the last two days it was traded, but not including the current day) and the difference between that moving average and the current price.
4. Using the FND data and an analytic function, return the number of stocks alphabetically before each stock (e.g. if "A" was the first company would be 0, AA would be 1, etc.) in 2010. Make sure to only include each name once. Feel free to include or exclude company names that begin with a number.[14]
5. Without using an analytic function answer the same question as above.

Main Problems

1. For each stock in 2010, return the largest average daily return ((close - open)/open) by the first letter of the symbol. In other words, there should be one row for each first letter of the ticker symbol. The dataset should return (a) the symbol which has the largest, (b) the total number of symbols which begin with that character and (c) the average daily return for the symbol.
2. Repeat the above, this time without using an analytic function. Make sure that you *aren't* joining on a float as joining on a floating variable will lead to uneven results.
3. For stocks in 2010, write a query which creates a dataset containing closing price, symbol, retdate and the nominal change between yesterday's stock (symbol) price and today. If there is a missing day then the nominal change should be missing (which is different from the above question).

[14] Comparisons of the form *string* $\leq$ *string* do alphabetical comparison. Also keep in mind that you can join using any conditional expression.

4. Return a dataset which contains symbol and the number of trading days that the price is within 10% of the max price for that year for each stock. For example, if the max price of a symbol is $100, then return the number of times that price of that stock is ≥ 90.
5. For each symbol return the number of days it took to reach its maximum closing price for that year. If a stock is not traded on a day, then that should *not* count toward the total days. Note that there should be one row per symbol in the final dataset.
6. Repeat the previous question without using any analytic functions.
7. For each stock symbol return the number of days it took to reach its maximum closing price for that year. If a stock is not traded on a day, then it should count toward the total days.

Extra Problems

1. In the Transaction data, what percentage of users, who start by purchasing a Unit end up Subscribing?
2. What percentage of users, in the transaction data, purchase both a Unit and a subscription?
3. What is the average amount of time between Unit Purchases?
4. Calculate the 25, 50 and 75 percentile of the amount of revenue generated in the first six months (per user). Only include those users who made their first purchase more than six months ago. Make sure that this query moves with time: if I run this query next month it should return updated data.

DRAFT

18 HW #8A: SQL Window Functions

Using only the functions and syntax that we have learned in class, please provide a query to answer the following questions. If a dataset is not specified, please use the 2010 dataset. **Do not create any tables or views.**

Before beginning the assignment, *please read the data dictionary to better understand the data.* When doing so, keep an eye on data types for different columns as well as table organization.

- If no year information is provided for a financial question, assume 2010.
- If the query returns a significant number of rows, please only copy a few rows in your response.

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

DRAFT

First Five

1. Write a query which returns, for stocks in 2010, the symbol, the date and the cumulative sum of traded volume for that stock from the start of the year to that date, including that date.
2. Repeat the above without an analytic function.
3. Write a query which returns, for stocks in 2010, the symbol, the date and the cumulative sum of traded volume for that stock from the start of the year to that date, *not* including that date.
4. Repeat the above without an analytic function.
5. Write a query which returns, for the stocks in 2010, the symbol, the date, and the moving average of the last five days (including the current date) of closing prices for that stock.

Main Problems

1. Write a query which returns, for stocks in 2010, the symbol, the date and the cumulative sum of traded volume for that stock from the start of the *current month*, including that date.
2. Repeat the above without an analytic function.
3. Write a query which returns, for stocks in 2010, the symbol, the date, the ratio of that days stock closing price to the stock's closing price on the first day that the stock is traded that year ($\frac{current_price}{first_price}$)
4. Write a query which returns, for stocks in 2010, the symbol, the date and the difference between the max closing price that the stock achieves in 2010 and the current day's closing price.
5. What is the median closing price for all stocks on January 4th, 2010?
6. For stocks in 2010, write a query which returns the closing price, symbol, retdate and the nominal change between yesterday's closing price and today's opening price. Ignore holes in the data, so that if the stock misses a day the change in price is from the last time listed.
7. Write a query which returns, for stocks in 2010, a set of unique symbols and a column which is the alphabetical rank (e.g. so "A" should be 1, "AA" should be 2, etc.)

19 HW #8B: Pandas Window Functions

In order to receive full credit, please turn in a document which is python code containing what would be run to return the data asked.

The queries below rely on information from the stock return data. To load the data use the following commands. **Note: these load retdate as a date, rather than a string**

The best approach to learning from these problems is to complete them using pen and paper, working by yourself and then using your group to double check your results. The First Five problems provide a short overview of the core concepts in the assignment, so make sure that you understand them. The Main Problems section contains questions which range from easy to very difficult. Remember to don't get stuck! If a problem is taking a long time or is too difficult, *use your group!*

```
## Initial Information
import pandas as pd
import numpy as np

df2010D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2010.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

df2011D = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/2011.tdf',
        sep='\t', engine='python', names=['symb', 'retdate', 'opn', 'high', 'low',
        'cls', 'vol', 'exch'], parse_dates=['retdate'])

dffnd = pd.read_csv('/Users/ncross/git/sqlnotes/newserver/data/fnd.tdf',
        sep='\t', engine='python', names=['gvkey', 'datadate', 'fyear', 'indfmr',
        'consol', 'popsrc', 'datafmt' , 'tic', 'cusip', 'conm', 'fyr', 'cash', 'dp',
        'ebitda', 'emp', 'invt', 'netinc', 'ppent', 'rev' , 'ui', 'cik'])

dfMTA = pd.read_csv('../sql-data/raw_data/mta/MTA_Hourly.tdf', sep='\t',
        engine='python', names=['plaza', 'mtadt', 'hr', 'direction', 'vehiclesez',
        'vehiclescash'])

dfTrans = pd.read_csv('../sql-data/raw_data/soapData.tdf', sep='\t',
        engine='python', names = ['orderid', 'userid', 'trans', 'type', 'local',
        'trans_dt', 'units', 'coupon', 'months', 'amt' ])
```

First Five

1. For each stock in 2010, return the original dataset as well as a column ("newcol") which is the 3 day moving average of the closing price (making sure to include the current closing price in the average).
2. For each stock in 2010, return the original dataset as well as a column ("newcol") which is the 3 day moving average of the closing price (making sure to exclude the current closing price in the average).
3. Calculate the average correlation between closing price and volume. To do this calculate the correlation for each stock and then take the average over all stocks. This should return a single number.
4. For each stock in 2010, calculate the percentage of the historical max closing price, up to (but not including) that point, that the current closing price is. Note that whenever a new max closing price is achieved the percent would be greater than 100.
5. Using `stack` or `unstack` create a DataFrame which is one row per symbol with columns for each month in 2010. The values in those columns should be the average closing price.

Main Problems

1. Using `stack` or `unstack` create a DataFrame which is one row per symbol with columns for each month in 2010. There should be multiple columns for each month, one for the average closing price, one for the average volume and one for the maximum volume (37 Columns: Symbol, Jan-Dec for average closing price, Jan-Dec for average volume and Jan-Dec for maximum volume). The DataFrame returned should not have a row index.

2. Using `stack` or `unstack` create a DataFrame which is one row per symbol with 12 columns which should be the cumulative volume for that month (including that month) over the entire year of 2010. E.g. This should be the running sum, but then accumulated per month.

3. Using `stack` or `unstack` create a DataFrame which is one row per symbol with columns for each month in 2010 *and* 2011. The values in those columns should be the average closing price for that month.

4. For stocks in 2010, write a query which creates a dataset containing closing price, symbol, retdate and the nominal change between yesterday's closing price and today's opening price. Ignore holes in the data, so that if the stock misses a day the change in price is from the last time listed.

DRAFT

20 BART Project

The objective of this assignment is to create a Python script which loads the BART data into your local database. In order to receive full credit on this assignment you will need to write a Python Script which takes the raw Excel files and loads the "core" ridership data.[15]

The table that holds the data should have the following form:

```
CREATE TABLE cls.bart (
        mon int
        , yr int
        , daytype varchar(15)
        , start varchar(2)
        , term varchar(2)
        , riders float
);
```

Requirements:

- Your code should be callable from a *single* function. While you can have multiple functions (or use objects), the entire script should be run via a single command.
- The code should be in a single text file. No notebooks.
- The code should be robust to being run more than once. If the code is run twice in a row, it should not break, crash or duplicate the data in the database.
- For older time periods the clipper/fastpass data may be broken out, just use the main data and ignore the clipper data.
- You should assume that the code is going to be run on a clean computer. Any implied file structure or libraries that need to be present should be removed.
- The overall structure of the program should be as follows:
  1. Assume that all of the zip files are in a single directory (*dataDir*), which is taken as a parameter in the function.
  2. The code should unzip the files into a directory (*tmpDir*).
  3. The code should process the Excel files, extracting necessary data and reshaping it so that it can be loaded.
  4. A table should be created in your database.
  5. The clean, reshaped and standardized data should then be copied in.
- Things that you will need to standardize:
  - The format for year and month changes over time. Your code should standardize these changes.
  - The number of stations changes over time. If a particular file does not have a station, there is no need to add it.
  - The daytypes ("Weekday", "Saturday" and "Sunday") change their names throughout the data. Make sure that they are standardized. You can ignore the phrase "adjustments." The data was calculated the same way over the entire time period.

[15] For more information about the BART data, please look at `https://www.bart.gov/about/reports/ridership`

DRAFT

- You can assume that the schema has already been created, but you will need to handle the table creation yourself.
- You need to verify that you only load the appropriate files. In other words, make sure to either track files through the process or delete everything within the temp directory before placing files in it.
- Don't use Pandas. It's janky.
- Note that the data in the Excel spreadsheets is presented in a wide format – each column represents the average exits for a particular station. The target table ("cls.bart") is long, not wide; the data will require reshaping before it is copied in.
- The function *ProcessBart* should be called in the following manner:

  ```
  ProcessBart( tmpDir, dataDir, SQLConn=None, schema='cls', table='bart')
  ```

  the parameters of the function:

  - *tmpDir*: Directory where the unzipped files should be stored.
  - *dataDir*: Directory where the zipped files are stored.
  - *SQLConn*: Psycopg2 connection.
  - *schemaName*: The schema where the data should be loaded.
  - *tableName*: The table where the data should be loaded.
- By "core" data, I mean the Weekday, Saturday and Sunday data. Note that in many of the files there are secondary tables or sheets. For example, in the January 2011 data on the "Weekday OD" sheet, the only information that should be copied is B3:AR45.
- Think hard about what code can be repeated and what code should be put into loops or turned into functions. Needlessly repetitive code will be penalized.

Hints:

- Libraries that I used when writing this code:
  - Psycopg2
  - glob
  - xlrd
  - zipfile
  - os
  - shutil

  You can use any other library that can be installed via pip.
- Think hard about what needs to be standardized between years. The difficult part of this code is creating a data structure that allows you to iterate over the years smoothly.
- Please use psycopg2 in order to interface with the database.
- In my code, the create table (using psycopg2) looks like:

DRAFT

```
## Load into DB
SQLCursor = SQLConn.cursor()
SQLCursor.execute("""
  CREATE TABLE %s.%s
  (
  mon int
  , yr int
  , daytype varchar(15)
  , start varchar(2)
  , term varchar(2)
  , riders float
  );""" %(SchemaName, TableName))
  SQLCursor.execute("""COPY %s.%s FROM '%s' CSV;"""
          % (SchemaName, TableName, tmpDir + 'toLoad.csv'))
  SQLConn.commit()
```

- Note that I created a CSV file, "toLoad.csv" inside *tmpDir* to put the formatted and reshaped data.
- Finally, when I grade this code, I am going to download your python script to my personal computer. I will then append the following to your script and run it.

  ```
  LCLconnR = psycopg2.connect("dbname='ncross' user='ncross'
           host='localhost' password='XXX'")

  ProcessBart( '\home\ncross\tmp\', '\home\ncross\BART\',
           SQLConn=LCLconnR, schema='cls', table='bart')
  ```

  Assuming that your code runs (and I hope it does), I will then run 3-5 SQL queries on the resulting data to verify that it loaded completely and correctly.
- I will also be reading over the code itself. While I do not expect you to be Python wizards, I do expect you to be able to code efficiently. This means using loops, functions and variables to create well-written code that also contains comments to include readability.
- Please make sure that the code removes files from the temp directory before trying to load or only works on specific files that you choose. If a file is in that directory that you do not expect it should not cause your code to fail.

DRAFT

21 HW #5AO: Info Schema and Price-Volume Relationship [TBD]

First Five

Using the information schema answer the following questions:

1. Write a query which returns the count of data types (int, float, etc.) of each columns in the stocks schema.
2. Write a query which returns the number of distinct column types in the entire database.
3. Write a query which returns 3 columns: schema name, column data type, and the number of columns in that schema of that column type.
4. Rewrite the above query in a wide-format. Each row should represent a single schema.
5. Create a pie chart of the above information for the schema "information_schema". Which data format (wide or long) did you use?

In the following exercise, we will investigate the relationship between the dollar volume of shares traded and the returns of that company. Exploring the relationship between dollar volume and return:

1. Write a query which returns the return rounded to the nearly thousandth of a percent while dealing with any data issues. Return the data in hundredths, so if the return is .037123, 3.7 should be returned. Include the dollar volume of stocks traded that day, rounded to the nearest 1,000. Also, only take a 1/16 sample using the following where statement:

   ```
   where md5( permno::varchar(100) ) like '0%'
   ```

2. Create a scatter plot of your rounded returns vs. the rounded dollar volume.
3. Run simple linear regression on the rounded returns vs. the rounded dollar volume and report the results. Do you believe that there is a relationship between trading size and dollar volume traded?
4. Recreate the scatter plot making sure to remove days with less than 250 million shares traded and only include returns between -10 and 10. Did the pattern change?
5. Run simple linear regression on the rounded returns vs. the rounded dollar volume and report the results for the sample of more than 250 million shares and returns between -10 and 10. Do you believe that there is a relationship between trading size and volume traded?
6. Using only the SUM, AVG and COUNT aggregate functions, compute the variance of both the rounded volume and the rounded returns of the sample.
7. The problem below is from the analytic function lecture and should be incorporated in to the LTV estimate. Write a query which returns the following information. Cohort should be defined monthly.
   (a) For each *complete* month, calculate the percentage revenue generated, per cohort, when compared to the previous month. For example, if Month #2 after first purchase the amount of revenue generated is equal to $12,755.54 and the amount of money generated in Month #1 after purchase is equal to $24,885.32 then return $\frac{12{,}755.54}{24{,}885.32} = .513376$[16]
   (b) Average this over all the cohorts with complete month data. Be careful to only consider dates that are complete from both the start and end of the table.
   (c) This should return a set of month-over-month multipliers that could be used to estimate the expected revenue generated from a new cohort. Explain how these numbers could be used to

[16] A complete month is one that is 100% in the data. For example, if a company launches on January 12, 2017 then January is not a complete month. Similarly, if today's date is September 19th, then September is not a complete month.

DRAFT

estimate the lifetime value of a customer (in their first year). E.g. If a customer generated \$1 of revenue in their first month, what would you do with those multipliers to estimate the lifetime value?

DRAFT

DRAFT

Appendix D

Example Exams

This textbook is used for a variety of different courses and course formats. The exams included in this appendix reflect this diversity. In order to study from these exams, keep in mind that the amount of time given and material covered may be different.

DRAFT

1 2023 CAPP Databases Final A

This exam was given to Master's level students in the University of Chicago CAPP program in the Spring of 2023. There were two versions of the exam – this is version "A".

The following tables contains information about Uber Eats drivers, their deliveries and reviews. Keep in mind that (a) not all deliveries will have reviews. A delivery can have, at most, one review. Not all drivers will have deliveries. When a driver first signs up they will not have any deliveries or reviews

- Only use syntax covered in class. Do not create any views.
- Interpret all inequalities as strict unless explicitly stated.
- If there is no specified return format (DataFrame/Series/etc.) than any format will be accepted.
- If you provide more than one answer, the lower of the two scores will be counted.
- Any two columns with the same name can be assumed to match.
- Columns in the *drivers* table / DataFrame:
  - **driver_id:** The ID of the driver (INT, UNIQUE, NOT NULL).
  - **state:** The state that the driver lives in (STRING, NOT NULL).
  - **age:** The age of the driver in years (INT, NOT NULL).
- Columns in the *reviews* table / DataFrame:
  - **del_id:** The ID of the delivery being reviewed (INT, NOT NULL).
  - **review:** The review score on a 1 (worst) to 5 (best) scale (INT, NOT NULL).
- Columns in the *delivery* table / DataFrame:
  - **del_id:** The unique ID associated with the delivery (INT, UNIQUE, NOT NULL).
  - **del_date:** The date that the delivery occurred (DATE, NOT NULL).
  - **driver_id:** The ID of the driver (INT, NOT NULL).
  - **car:** A flag (1/0) for if the driver used a car to make the delivery (INT, NOT NULL).
  - **length:** The distance that the delivery took, in km (FLOAT, NOT NULL).

driver_id	state	age
1	CA	27
2	MN	37
3	CA	28
4	CA	32

Drivers (1,234 Rows)

del_id	review
1	5
23	4
35	4
45	1

Reviews (980 Rows)

del_id	del_date	driver_id	car	length
1	1-1-2012	45	0	1.25
2	12-23-2012	45	1	23.45
3	7-6-2013	112	1	11.17
4	5-5-2014	1125	0	.75

Deliveries (14,365 Rows)

SQL Section

1. Write a query which returns the 7 oldest drivers (*driver_id* only) from the state of Michigan ("MI").

DRAFT

```
select driver_id
from drivers
where state = 'MI'
order by age desc
limit 7
```

2. Write a query which returns three columns: (1) the state, (2) the total number of *drivers* (count) from that state (only including drivers who have one or more deliveries) and (3) the total number of *deliveries* (count) from that state. This should return one row per state.

```
select
        drivers.state
        , count( distinct drivers.driver_id)
        , count( deliveries.del_id)
from
        drivers
join
        deliveries
using( driver_id)
group by state
```

3. Write a query which returns one row per delivery and four columns. The first column should be the state of the driver, the second should be the *driver_id*, the third should be *del_id* and the fourth should be the total length that the driver has travelled up to and including that delivery (cumulative sum of *length*). Make sure that the cumulative sum is calculated by the date of the delivery from earliest to latest. If a driver does not have any deliveries they should *not* be included in the results.

```
select
        driver_id
        , del_id
        , state
        , sum( length) over(
                partition by driver_id
                order by del_dt asc
                rows between unbounded preceding and current row
                ) as cum_sum
from
        drivers
join
        deliveries
using( driver_id )
```

4. Write a query which returns two rows and two columns. The first column should be state and the second should be the number of 5-star reviews for deliveries from that state. Only include Michigan ("MI") and Pennsylvania ("PA").

```
select
        state,
        count( case when review = 5 then 1 else null end ) as num_five_stars
from
        (select * from drivers where state in ('MI', 'PA')) as lhs
left join
        deliveries
        using( driver_id)
left join
        review
        using( del_id)
group by state
```

5. We want to return the average length of deliveries depending on if the driver used a car or not. Write a query which returns two rows and two columns. The first column should be if the delivery person used a car or not (the *car* column 1/0 flag) and the second column should be the average length of a delivery with that particular flag value. Only include those observations from the year 2012.

```
select
        car,
        avg( length)
from
        deliveries
where
        date_part('year', del_date) = 2012
group by 1
```

6. Please return one row and two columns. The first column should be the average length of deliveries when a car is used ($car = 1$) and the second column should be the average length of a delivery when a car is not used ($car = 0$). We want to calculate this on all deliveries from February in any year. Note that this is similar to the last problem, but the data shape and date filters are different.

```
select
        avg( case when car = 0 then length else null end) as car_0_avg
        , avg( case when car = 1 then length else null end) as car_1_avg
from
        deliveries
where
        date_part('month', del_date) = 2;
```

7. What state (*state* only) has the most drivers?

```
select state
from drivers
group by 1
order by count(1) desc
limit 1;
```

8. What was the longest (by *length*) non-car ($car = 0$) delivery (*del_id* only)?

```
select
        del_id
from
        deliveries
where car = 0
order by length desc
limit 1;
```

9. We call drivers who have ever done a delivery of more than 60 km a “long-driver”. What is the average review score for “long-drivers”? This should include *all* deliveries from “long-drivers”, even those less than 60 km. This should return only a single row and column.

```
select
        avg( review )
from
        deliveries
left join
        reviews
using(del_id)
where
        driver_id in (select distinct driver_id from deliveries where length > 60)
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that DataFrames named *drivers*, *deliveries* and *reviews* are already loaded. Unless otherwise specified you may return either a Series or DataFrame.

1. Return a DataFrame which has two columns and a row for each state. The first column should be the state and the second should be the number of deliveries completed by drivers from that state.

```
pd.merge( drivers, deliveries, on='did', how='left')
        .groupby('state', as_index=False)
        .agg({'del_id' : ['count']})

-- Since this is number of jobs it could be inner or left or outer join
```

2. Return a DataFrame with two rows and two columns. The first column should be a flag which takes one of two values: “LT10” and “MT15”. The second column should be the average review for deliveries which are (strictly) “Less Than 10 km” and “More Than 15 km” , respectively. In other words, one row should contain “LT10” and the average review for deliveries which are less than 10 km and the other row should contain “MT15” and the average review for deliveries which are more than 15 km.

DRAFT

```
mrg = pd.merge( deliveries, reviews, on='del_id', how='inner')

mrg = (mrg
       .loc[(mrg.loc[:, 'length'] < 10) | (mrg.loc[:, 'length'] > 15), :]
      )

mrg.loc[:, 'flag'] = 'LT10'
mrg.loc[(mrg.loc[:, 'length'] >15), 'flag'] = 'MT15'

mrg.groupby('flag', as_index=False).agg({'review' : ['mean']})
```

3. We call drivers who have ever done a delivery of more than 60 km a "long-driver". What is the average review score for "long-drivers"? This should include *all* deliveries from "long-drivers", even those less than 60 km. This should return a single value (can be in a DataFrame, in a Series or as a number).

```
driver_id_lst = deliveries.loc[(deliveries.loc[:, 'length'] > 60), 'driver_id'].drop_duplicates()

mrg = pd.merge( reviews, deliveries, on='del_id', how='inner')
mrg.loc[ (mrg.loc[:, 'driver_id'].isin( driver_id_lst), 'review'].mean()
```

4. What was the longest (by length) non-car ($car = 0$) delivery (del_id only)?

```
deliveries.loc[ (deliveries.loc[:, 'car'] == 0), :].nlargest(1,'length').loc[:, 'del_id']

OR

(deliveries
        .loc[ (deliveries.loc[:, 'car'] == 0), :]
        .sort_values('length', ascending=False)
        .loc[:, 'del_id']
)
```

5. Which year had the largest number of deliveries (count)?

```
(deliveries
        .assign(yr = deliveries.loc[:, 'del_date'].dt.year)
        .groupby( yr, as_index=False)
        .agg( {'del_id' : ['count']})
        .nlargest(1, ('del_id', 'count'))
        .loc[:, 'yr']
)
```

6. How many drivers are from California ("CA")?

```
drivers.loc[ (drivers.loc[:, 'state'] == 'CA'), :].shape[0]

Lots of ways to do this one.
```

7. Which date (*del_date*) had the largest number of 3-star deliveries (count)?

```
mrg = pd.merge( deliveries, reviews, how='left', on='del_id')

(mrg
        .loc[ (mrg.loc[:, 'review'] == 3), :]
        .groupby( 'del_date', as_index=False)
        .agg( {'del_id' : ['count']})
        .nlargest(1, ('del_id', 'count'))
        .loc[:, 'del_date']
)
```

8. Return all drivers (*driver_id* only) from Texas ("TX") who are 32 years old as a *DataFrame*.

```
drivers.loc[ (drivers.loc[:, 'state'] == 'CA') & (drivers.loc[:, 'age'] == 32), ['driver_id']]
```

DRAFT

2 2023 CAPP Databases Final B

This exam was given to Master's level students in the University of Chicago CAPP program in the Spring of 2023. There were two versions of the exam – this is version "B".

The following tables contains information about a large painting company. The company has many painters who do jobs and then get reviews on those jobs. A job can, at most, have **one** review. Not all painters will have jobs (when they first start there is some time before they are assigned a job). A painter can have multiple jobs as most jobs last only a day or two and every job will be in the database.

- Only use syntax covered in class. Do not create any views.
- Interpret all inequalities as strict unless explicitly stated.
- If there is no specified return format (DataFrame/Series/etc.) than any format will be accepted.
- If you provide more than one answer, the lower of the two scores will be counted.
- Any two columns with the same name can be assumed to match.
- Columns in the *painters* table / DataFrame:
  - **painter_id:** The ID of the painter (INT, UNIQUE, NOT NULL).
  - **state:** The state that the painter works in (STRING, NOT NULL).
  - **work_exp:** The number of years of work experience (INT, NOT NULL).
- Columns in the *jobs* table / DataFrame:
  - **job_id:** The unique ID associated with the job (INT, UNIQUE, NOT NULL).
  - **job_date:** The date that the job occurred (DATE, NOT NULL).
  - **painter_id:** The ID of the painter (INT, NOT NULL).
  - **sprayer:** A flag (1/0) for if the painter used a sprayer or not (INT, NOT NULL).
  - **paint:** The amount of paint used, in gallons (FLOAT, NOT NULL).
- Columns in the *reviews* table / DataFrame:
  - **job_id:** The ID of the job being reviewed (INT, NOT NULL).
  - **review:** The review score on a 1 (worst) to 5 (best) scale (INT, NOT NULL)

painter_id	state	work_exp
1	CA	0
2	MN	12
3	CA	4
4	CA	11

Painters (1,234 Rows)

job_id	job_date	painter_id	sprayer	paint
1	1-1-2012	45	0	1.25
2	12-23-2012	45	1	23.5
3	7-6-2013	112	1	11.25
4	5-5-2014	1125	0	.75

Jobs (14,365 Rows)

job_id	review
1	5
23	4
35	4
45	1

Reviews (980 Rows)

DRAFT

SQL Section

1. Write a query which returns the 11 painters (*painter_id*) with the most experience (largest *work_exp*) from Hawaii ("HI").

```
select painter_id
from painters
where state = 'HI'
order by work_exp desc
limit 11
```

2. Write a query which returns three columns: (1) the state, (2) the total number of *painters* (count) from that state (only including painters who have one or more jobs) and (3) the total number of *jobs* (count) from that state. This should return one row per state.

```
select
        state
        , count( distinct painters.painter_id)
        , count( jobs.job_id)
from
        painters
join
        jobs
using( painter_id )
group by state
```

3. Write a query which returns one row per job and four columns. The first column should be the state of the painter, the second should be the *painter_id*, the third should be *job_id* and the fourth should be the total amount of paint that the painter has used up to and including that job (cumulative sum of *paint*). Make sure that the cumulative sum is calculated by the date of the job from earliest to latest. If a painter does not have any jobs they should *not* be included in the results.

```
select
        painter_id
        , job_id
        , state
        , sum( paint ) over(
                partition by painter_id
                order by job_dt asc
                rows between unbounded preceding and current row
                ) as cum_sum
from
        painters
join
        jobs
using( painter_id )
```

4. Write a query which returns two rows and two columns. The first column should be state and the second should be the number of 5-star reviews for jobs from that state. Only include Alaska ("AK") and Hawaii ("HI").

```
select
        state,
        count( case when review = 5 then 1 else null end ) as num_five_stars
from
        (select * from painters where state in ('AK', 'HI')) as lhs
left join
        jobs
        using( painter_id)
left join
        review
        using( job_id)
group by state;
```

5. We want to return the average amount of paint used depending on if the painter used a sprayer or not. Write a query which return two rows and two columns. The first should be if the painter used a sprayer or not (the *sprayer* column 1/0 flag) and the second should be the average amount of paint used. Only include those observations from the year 2014.

```
select
        spray,
        avg( paint )
from
        jobs
where
        date_part('year', del_date) = 2014
group by 1
```

6. Please return one row with two columns. The first column should be the average amount of paint used when a sprayer is used ($sprayer = 1$) and the second column should be the average amount of paint used if a sprayer is not used ($sprayer = 0$). We want to calculate this on all jobs from July in any year. Note that this is similar to the last problem, but the data shape and date filters are different.

```
select
        avg( case when sprayer = 0 then paint else null end) as spray_0_avg
        , avg( case when sprayer = 1 then paint else null end) as spray_1_avg
from
        jobs
where
        date_part('month', del_date) = 7;
```

7. What state (state only) has the most painters?

```
select state
from painters
group by 1
order by count(1) desc
limit 1;
```

8. What was the largest (used the most paint) non-sprayer ($sprayer = 0$) paint job? Return the *job_id* only.

```
select
        job_id
from
        jobs
where sprayer = 0
order by paint desc
limit 1;
```

9. We call painters who have ever done a job of more than 25 gallons "large-scale" painters. What is the average review for "large-scale" painters? This should include *all* jobs from "large-scale" painters, even those jobs which are less 25 gallons. This should return only a single row and column.

```
select
        avg( review )
from
        jobs
left join
        reviews
using(job_id)
where
        painter_id in (select distinct painter_id from jobs where paint > 25)
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that DataFrames named *painters*, *jobs* and *reviews* are already loaded. Unless otherwise specified you may return either a Series or DataFrame.

1. Return a DataFrame which has two columns and a row for each state. The first column should be the state and the second should be the number of jobs completed by painters from that state.

```
pd.merge( painters, jobs, on='painter_id', how='left')
        .groupby('state', as_index=False)
        .agg({'job_id' : ['count']})

-- Since this is number of jobs it could be inner or left or outer join
```

2. Return a DataFrame with two rows and two columns. The first column should be a flag which takes one of two values: "less3" and "greater20". The second column should be the average review for jobs which are (strictly) "less than 3 gallons" and "greater than 20 gallons" , respectively. In other words, one row should contain "Less3" and the average review for jobs which are less than 3 gallons and the other row should contain "greater20" and the average review for jobs which are more than 20 gallons.

```
mrg = pd.merge( jobs, reviews, on='job_id', how='inner')

mrg = (mrg
       .loc[(mrg.loc[:, 'paint'] < 3) | (mrg.loc[:, 'length'] > 20), :]
      )

mrg.loc[:, 'flag'] = 'less3'
mrg.loc[(mrg.loc[:, 'paint'] > 20), 'flag'] = 'greater20'

mrg.groupby('flag', as_index=False).agg({'review' : ['mean']})
```

3. We call painters who have ever done a job of more than 25 gallons "large-scale" painters. What is the average review for "large-scale painters? This should include *all* jobs from "large-scale" painters, even those jobs which are less 25 gallons. This should return a single value (can be in a DataFrame, in a Series or as a number).

```
painter_id_list = jobs.loc[(jobs.loc[:, 'paint'] > 25), 'driver_id'].drop_duplicates()

mrg = pd.merge( reviews, jobs, on='driver_id', how='inner')
mrg.loc[ (mrg.loc[:, 'painter_id'].isin( painter_id_list), 'review'].mean()
```

4. What was the largest (most paint used) non-spray (*sprayer* = 0) job (job_id only)?

```
jobs.loc[ (jobs.loc[:, 'sprayer'] == 0), :].nlargest(1,'paint').loc[:, 'job_id']

OR

(job
        .loc[ (job.loc[:, 'sprayer'] == 0), :]
        .sort_values('paint', ascending=False)
        .loc[:, 'job_id']
)
```

5. Which year had the largest number of jobs (count)?

```
(job
        .assign(yr = job.loc[:, 'job_date'].dt.year)
        .groupby( yr, as_index=False)
        .agg( {'job_id' : ['count'])
        .nlargest(1, ('job_id', 'count'))
        .loc[:, 'yr']
)
```

6. How many painters are from New Mexico ("NM")?

```
painters.loc[ (painters.loc[:, 'state'] == 'CA'), :].shape[0]

Lots of ways to do this one.
```

7. Which date (*job_date*) had the largest number of 2-star jobs (count)?

```
mrg = pd.merge( jobs, reviews, how='left', on='job_id')

(mrg
        .loc[ (mrg.loc[:, 'review'] == 2), :]
        .groupby( 'job_date', as_index=False)
        .agg( {'job_id' : ['count']})
        .nlargest(1, ('job_id', 'count'))
        .loc[:, 'job_date']
)
```

8. Return all painters (*painter_id* only) from California ("CA") who have 12 years of experience as a *DataFrame*.

```
painters.loc[ (painters.loc[:, 'state'] == 'CA') & (painters.loc[:, 'age'] == 32), ['painter_id']]
```

DRAFT

3 2023 CAPP Databases Midterm A

This exam was given to Master's level students in the University of Chicago CAPP program in the Spring of 2023. There were two versions of the exam – this is version "A".

The following table contains information about campers at a camp. You can assume each name uniquely defines a camper and that a camper only appears once in the table.

- **name:** The name of the camper (string, NOT NULL).
- **age:** The camper's age, in years (integer, NOT NULL).
- **hgt:** The height of the camper (in centimeters) (float, NOT NULL).
- **state:** The state that the camper is from (string, NOT NULL).
- **program:** The specific program (elective) that the camper choose. You can assume this is all lower case (string, NOT NULL).
- **amt_paid:** The amount they paid to attend camp (float, NOT NULL).
- **allergy:** If a camper has a food allergy (if Null that means no allergy). You can assume this is all lower case. There will be only a SINGLE allergy listed (string, HAS NULLS).
- The name of the table / DataFrame is **camper**. No need to use a schema or load the DataFrame.
- Only use syntax covered in class.
- Interpret all inequalities as strict unless explicitly stated.

Figure D.1: *camper* Table: 12,345 Rows

name	age	hgt	state	program	amt_paid	allergy
Ringly Roberson	7	121.0	NY	basketball	987.54	
Crash Bandicoot	11	144.5	MS	basket-weaving	1128.75	peanuts
Alligator Reynolds	8	129.0	PA	skateboarding	585.46	

DRAFT

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Using SQL, write a query which returns the names (name only) of the top-8 shortest campers who are 10 years old.

```
SELECT
        name
FROM
        campers
where age = 10
order by hgt asc limit 8;
```

2. Using SQL, write a query which returns all campers (name only) who are younger than 10 years old and are either from New Jersey ('NJ') or Wyoming ('WY'). Only include those campers who do NOT have a food allergy.

```
select name from campers where
        state in ('NY', 'AL')
        and age < 10
        and allergy is null;
```

3. Write an SQL query which returns the number of campers from each state who have food allergies. This should be two columns: one with the state and the other with the number of campers from that state who have food allergies.

```
SELECT
        state, count(1) as ct
from
        campers
where injury is not null
group by 1;
```

4. Write an SQL query which returns all rows and columns for campers who are taking "basketball" as their program (you can assume that all programs are lowercase). Sort them from tallest to shortest.

```
select * from campers
where program = 'basketball'
order by hgt desc;
```

5. We define an "allergy impacted" program as one with 10% (or more) of the campers in that program having a food allergy or any kind. Write an SQL query which returns a list of programs which are "allergy impacted". This should return 1 column with a list of "allergy impacted" programs.

```
select program from
        (select program
                , sum( case when allergy is not null then 1 else 0 end)::float / sum(1) as rat
        from campers
        group by 1 ) as innerQ
where rat >= .1
```

6. We calculate the age-adjusted height ("AAH") by taking a campers height and dividing it by their age squared ($\frac{height}{age^2}$). Write a query which returns all rows and three columns: age-adjusted height, name and program.

```
select hgt /age / age as aah, name, program
from campers;
```

7. Using SQL, write a query which returns three columns: name, program, and AAH_Flag. AAH_Flag should be equal to 0 if the AAH is less than or equal to 1, 1 if the AAH is greater than 1 and less than or equal to 3 and 2 otherwise. AAH is defined in the previous problem.

DRAFT

```
select
        name, program
        , case
                when hgt / age / age  <= 1 then 0
                when hgt / age / age <= 3 then 1
                else 2 end as AAH_Flag
from
        campers;
```

8. If a person's AAH is greater than or equal to 3 they are defined as "tall". Write a query which returns the *percentage* of campers of each program who are tall. This should have two columns: program and percentage of the campers in that program who are tall.

```
select
        program,
        sum( case when hgt / age / age >= 3 then 1 else 0 end )::float / count(1) as pctTall
from
        campers
group by 1;
```

9. Using SQL, write a query which returns one row and two columns. The first column should be the number of campers who are 10 years old (exactly) and signed up for the "basketball" program (call this column bb10). The second column should be the number of campers who are 7 years old (exactly) who are signed up for the "skateboarding" program (call this column sb7). You can assume that all programs are lower case.

```
select
        sum( case when program = 'basketball' and age = 10 then 1 else 0 end) as bb10
        , sum( case when program = 'skateboarding' and age = 7 then 1 else 0 end) as sb7
from campers;
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that a DataFrame named *campers* is already loaded. If a specific output is not specified you can return anything (DataFrame/Series/List/Array/etc.)

1. Using Pandas, return the name (as a Series) of the top-8 shortest campers who are 10 years old.

```
campers.loc[(campers.loc[:, 'age'] == 10), :].nsmallest(8, 'hgt').loc[:, 'name']
```

OR:

```
(campers
        .loc[(campers.loc[:, 'age']== 10), :]
        .sort_values( 'hgt', ascending=True)
        .head(8)
        .loc[:, 'name']
)
```

2. Using Pandas, return a DataFrame with two columns: name and state. The dataset should only contain campers that are either (a) over 10 years and from Virginia ("VA") or (b) under 8 years old and from Michigan ("MI").

DRAFT

```
campers.loc[((campers.loc[:, 'age'] > 10) & (campers.loc[:, 'state'] == 'VA) )
        | ((campers.loc[:, 'age'] < 8) & (campers.loc[:, 'state'] == 'MI') )
        , ['name', 'state']]
```

3. Using Pandas, return a **DataFrame** which contains all campers (name only) who are younger than 10 years old and are either from New Jersey ('NJ') or Wyoming ('WY'). Only include those campers who do NOT have a food allergy.

```
campers.loc[(campers.loc[:, 'age'] < 10)
         & (campers.loc[:, 'allergies'].isna())
         & (campers.loc[:, 'state'].isin( ['AL', 'NY']))
         , ['name']]
```

4. Using Pandas, return all programs (this should be without duplicates) which have a camper who has an allergy to "peanuts". You can assume that all allergies in the table are lower case.

```
campers.loc[ (campers.loc[:, 'allergy'] == 'peanuts'), 'program'].unique()
```

5. Return all rows and columns for campers who are taking "basketball" as their program (you can assume that all programs are lower case). Sort the resulting DataFrame first by state (alphabetically) and then, within state, from tallest to shortest.

```
(campers.loc[ (campers.loc[:, 'program'] == 'basketball'), :]
         .sort_values( ['state', 'hgt'], ascending=[True, False])
         )
```

6. Please return a DataFrame which has all the original data and adds a column called "AAH" which is the age-adjusted-height (this is height divided by age squared, as in the previous problems).

```
campers.loc[:, 'aah'] = campers.loc[:, 'hgt'] / campers.loc[:, 'age'] / campers.loc[:,  'age']
```

7. Please return a DataFrame which has all the original data as well as adds a column called "hgt_flag" which is equal to 0 if the camper is greater than or equal to 140cm, 1 if they are greater than or equal to 110 and less than 140 and 2 otherwise.

```
campers.loc[:, 'hgt_flag'] = 2
campers.loc[ (campers.loc[:, 'hgt'] >= 140), 'hgt_flag'] = 0
campers.loc[  (campers.loc[:, 'hgt'] >= 110) & (campers.loc[:, 'hgt'] < 140), 'hgt_flag'] = 0
```

8. There was an error and students who were 10 years old all had their height recorded as 10 centimeters too high. Please return an updated version of the campers DataFrame which has this error fixed. Specifically the DataFrame should have all rows and columns, but the `hgt` column should have this error fixed.

```
campers.loc[ (campers.loc[:, 'age'] == 10), 'hgt'] = campers.loc[ (campers.loc[:, 'age'] == 10), 'hgt'] -10
```

DRAFT

4 2023 CAPP Databases Midterm B

This exam was given to Master's level students in the University of Chicago CAPP program in the Spring of 2023. There were two versions of the exam – this is version "B".

The following table contains information about workers applying to a temp agency for data entry positions. You can assume that each name uniquely defines a person and that a person only appears once in the table.

- **name:** The name of the person (string, NOT NULL).
- **exp:** Years of experience (integer, NOT NULL).
- **wpm:** The number of words per minute (wpm) the person types (float, NOT NULL).
- **state:** The state that the worker is from (string, NOT NULL).
- **degree:** Highest educational attainment. You can assume this is all lower case (string, NOT NULL).
- **wage:** Preferred hourly wage (float, NOT NULL).
- **certificate:** If the person has a special certificate, such as for dealing with health care or bank data. If Null, this means that the person has no certificate. You can assume this is all lower case. A person can, AT MOST, have a single certificate (string, HAS NULLS).
- The name of the table / DataFrame is **agency**. No need to use a schema or load the DataFrame.
- Only use syntax covered in class.
- Interpret all inequalities as strict unless explicitly stated.

Figure D.2: *agency* Table: 12,345 Rows

name	exp	wpm	state	degree	wage	certificate
Ringly Roberson	7	68.5	NY	high-school	18.00	
Crash Bandicoot	14	88.0	MS	ba	22.00	healthcare
Alligator Reynolds	4	72.25	PA	ms	17.5	

DRAFT

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Using SQL, write a query which returns the names (name only) of the 6 slowest typers (smallest wpm) who are from Pennsylvania ("PA").

```
SELECT
        name
FROM
        agency
where state = 'PA'
order by wpm desc
limit 6;
```

2. Using SQL, write a query which returns all workers (name and state) who have less than 10 years of experience, have no certificate and are either from Idaho ("ID") or California ("CA").

```
select name, state from agency where
        state in ('ID', 'CA')
        and exp < 10
        and certificate is null.
```

3. Write an SQL query which returns the number of workers from each state who have a certificate. This should be two columns: one with the state and the other with the number of workers from that state who have a certificate.

```
SELECT
        state, count(1) as ct
from
        agency
where certificate is not null
group by 1;
```

4. Write an SQL query which returns all rows and columns for workers whose degree is equal to "high-school". You can assume that all degrees are lower case. Sort the data by words per minute from highest to lowest.

```
select * from agency
where degree = 'high-school'
order by wpm desc;
```

5. We are analyzing degrees and certificates. We want to figure out which degrees have more than 20% of their workers with a certificate. Write an SQL query which returns one column. In this column should be a list of degrees where more than 20% of the workers with that degree have a certificate.

```
select degree from
        (select degree
                , sum( case when certificate is not null then 1 else 0 end)::float / sum(1) as rat
        from agency
        group by 1 ) as innerQ
where rat >= .2
```

6. We calculate the dollar-adjusted wpm ("DAWPM") by taking a worker's WPM, squaring it and dividing it by their wage (in pennies) ($\frac{wpm^2}{100 \cdot wage}$). Write a query which returns three columns: name, degree and the DAWPM.

```
select (wpm * wpm) / (100 * wage) as DAWPM, name, degree
from agency;
```

7. Write a query which returns three columns: name, degree, and DAWPM_Flag. DAWPM_Flag should be equal to 0 if the DAWPM is less than or equal to 3, 1 if the DAWPM is greater than 3 and less than or equal to 10 and 2 otherwise. DAWPM is defined in the previous problem.

DRAFT

```
select
        name, degree
        , case
                when (wpm * wpm) / (100 * wage)    <= 3 then 0
                when (wpm * wpm) / (100 * wage)   <= 10 then 1
                else 2 end as DAWPM_Flag
from
        agency;
```

8. If a person's DAWPM is greater than or equal to 3 they are defined as "hyper-efficient". Write a query which returns the *percentage* of workers of each degree who are hyper-efficient. This should have two columns: degree and percentage of the workers with that degree who are hyper-efficient.

```
select
        degree,
        sum( case when (wpm * wpm) / (100 * wage) >= 3 then 1 else 0 end )::float / count(1) as pctEff
from
        agency
group by 1;
```

9. Using SQL, write a query which returns one row and two columns. The first column should be the number of workers who have exactly 3 years experience and have a "ba" degree (call this column ba3). The second column should be the number of workers who have exactly 7 years of experience and have an "ms" degree (call this column ms7).

```
select
        sum( case when degree = 'ba' and exp = 3 then 1 else 0 end) as ba3
        , sum( case when degree = 'ms' and exp = 7 then 1 else 0 end) as ms7
from agency;
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that a DataFrame named *agency* is already loaded. If a specific output is not specified you can return anything (DataFrame/Series/List/Array/etc.)

1. Using Pandas, return the name (as a Series) of the top 6 fastest typers (largest wpm) who have 3 years of work experience.

```
agency.loc[(agency.loc[:, 'exp'] == 3), :].nlargest(6, wpm).loc[:, 'name']
```

OR:

```
(agency
        .loc[(agency.loc[:, 'exp'] == 3), :]
        .sort_values( 'wpm', ascending=False)
        .head(6)
        .loc[:, 'name']
)
```

2. Using Pandas, return a DataFrame with two columns: name and state. The dataset should only contain workers that are either (a) over 10 years experience and from Georgia ("GA") or (b) under

DRAFT

8 years experience and from Nevada ("NV").

```
agency.loc[((agency.loc[:, 'exp'] > 10) & (agency.loc[:, 'state'] == 'GA) )
        | ((agency.loc[:, 'exp'] < 8) & (agency.loc[:, 'state'] == 'NV') )
        , ['name', 'state']]
```

3. Using Pandas, return a **DataFrame** which contains all workers (name only) who have less than 9 years experience and are either from New Mexico ('NM') or Texas ('TX'). Only include those workers who have a certificate (any certificate).

```
agency.loc[(agency.loc[:, 'exp'] < 9)
        & ~(agency.loc[:, 'certificate'].isna())
        & (agency.loc[:, 'state'].isin( ['NM', 'TX']))
        , ['name']]
```

4. Using Pandas, return all states (this should be without duplicates) which have a worker with a "healthcare" certificate. You can assume that all certificate names are lower case.

```
agency.loc[ (agency.loc[:, 'certificate'] == 'healthcare'), 'state'].unique()
```

5. Return all rows and columns for workers who have a "doctorate" degree (you can assume that all degrees are lower case). Sort the resulting DataFrame first by state (alphabetically) and then, within state, from fastest to slowest wpm.

```
(agency.loc[ (agency.loc[:, 'degree'] == 'doctorate'), :]
        .sort_values( ['state', 'wpm'], ascending=[True, False])
        )
```

6. Please return a DataFrame which has all the original data and adds a column called "DAWPM" which is the dollar-adjusted words per minute (as in the other problems it is defined as words per minute squared divided by wages in pennies: $\frac{wpm^2}{100 \cdot wage}$).

```
agency.loc[:, 'dawpm'] = (agency.loc[:, 'wpm'] * agency.loc[:, 'wpm']) / (100.0 * agency.loc[:,  'wage'])
```

7. Please return a DataFrame which has all the original data as well as adds a column called "exp_flag" which is equal to 0 if the worker has less than 10 years experience, 1 if they have greater than or equal to 10 years and less than 20 years experience and 2 otherwise.

```
agency.loc[:, 'exp_flag'] = 0
agency.loc[  (agency.loc[:, 'exp'] >= 10) & (agency.loc[:, 'exp'] < 20), 'exp_flag'] = 1
agency.loc[ (agency.loc[:, 'exp'] > 20), 'exp_flag'] = 2
```

8. There was an error and workers from Maryland ("MD") had their WPM recorded as 5 too large. Please return an updated DataFrame which has this error fixed. Specifically the DataFrame should have all rows and columns, but the wpm column should have this error fixed.

```
agency.loc[ (agency.loc[:, 'state'] == 'MD'), 'wpm'] = agency.loc[ (agency.loc[:, 'state'] == 'MD'), 'wpm'] - 5
```

DRAFT

5 2017 SQL Final

This exam was given at the masters level in 2017 for a course just covering SQL. Students had two hours to complete it.

Use the following tables when answering the questions. The tables below consist of information from a manufacturing company's internal database system. This company makes children's toys by assembling *parts* into *items*.

- Columns with the same name can be assumed to be the same.
- The company takes *parts* and turns them into *items* which are then sold.
- **Unless stated above, all values are not null.**
- Parts table:
  - **PID:** The ID for a particular part. (int)
  - **Price:** The cost associated with the part. (float)
  - **Desc:** A short part description. (string)
  - You can assume that PID is unique to each row.
- Item table:
  - **IID:** The Item ID for a particular item. (int)
  - **Desc:** A short item description. (string)
  - **TPlus:** This is a 1 or 0, depending on if the toy is designed for children who are "Twelve Plus" years old. (int)
  - You can assume that IID is unique to each row.
- Assembly Table:
  - This table contains the "recipe" for making each item. It is a map between the parts and the items.
  - **IID:** This is the item ID of the item that is being assembled. (int)
  - **PID:** This is the PID of the part that is going into the item. (int)
  - **NoPart:** This is the quantity of each part that is required to make the item. (int)
- Sales Table:
  - This table contains information regarding the company's sales.
  - **IID:** This is the item ID for the item sold. (int)
  - **Rev:** This is the amount that the item was sold for. (float)
  - **CustID:** This is the ID for the customer. (int)
  - **SalesID:** This is the ID of the salesperson. (int)
  - **DT:** This is the date that the sales took place. (date)
- SP Table:
  - This table contains information regarding the salespeople in the company.

DRAFT

– Each row represents a salesperson.

– **SalesID:** This the ID of the salesperson. (int)

– **ST:** This is the state that the salesperson lives. If a state is NULL, that means the salesperson lives internationally. (string)

– **Name:** This is the salesperson's name.

– **Bonus:** This is equal to "H" or "L" for "High" or "Low" bonus structure. (string)

– You can assume that SalesID is unique to each row.

Table D.1: *Parts* Table, 4,525 Rows

PID	Price	Desc
1	11.99	Plastic Box (11 x 11)
2	12.85	8" Wheel
3	127.85	4" dowel

Table D.2: *Item* Table, 525 Rows

IID	Desc	TPlus
1	Small Wagon	0
2	Children's playhouse	0
3	Plastic Truck	1

Table D.3: *Assembly* Table, 15,225 Rows

IID	PID	NoPart
1	2	4
1	12	1
1	8	2

Table D.4: *Sales* Table, 45,258 Rows

IID	Rev	CustID	SalesID	Dt
12	65.73	1	12	01-03-2011
12	75.73	3	12	02-03-2011
48	265.04	2	17	01-05-2011
92	554.36	85	8	08-27-2011
115	18.18	92	22	08-11-2011

Table D.5: *SP* Table, 35 Rows

SalesID	ST	Name	Bonus
1	CA	Eldon Tyrell	H
2	PA	J.F. Sebastian	L
3		Roy Batty	L
4	CA	Rick Deckard	H

DRAFT

For the following questions write a query that returns the answer **and ONLY the answer.** All questions should be answered with a single SQL query, unless stated otherwise.

1. What are the top five salesperson (SalesID) in terms of *number* of items sold?

```
select SalesID
from sales
group by 1
order by sum(1) desc
limit 5;
```

2. How many salespeople are from California ("CA")?

```
select
        sum(1)
from
        SP
where st = 'CA';
```

3. Write a query which returns three columns and twelve rows. The first column should be month and should be the month that the sales took place, the second column should contain the number of items sold for the "Twelve Plus" audience and the third should contain the number of items sold for the not "Twelve Plus" audience. Assume that the sales table contains all information for the year 2011 (and no other year.

```
select
        date_part('month', dt) as mnth
        , sum( case when TPlus = 1 then 1 else 0 end ) as tplus
        , sum( case when TPlus = 0 then 1 else 0 end ) as tminus
from
        sales
left join
        item
using(iid)
group by 1;
```

4. Write a query which returns 5 columns. The first column is the state, the second column is the number of items sold to that state, the third is the number of items sold to that state which had a revenue greater than $1,000, the fourth column should the number of unique customers sold to that state and the final column should be the number of customers who spent more than $1,000 in a single transaction.

```
select
        lhs.st
        , count(1) as numitemsold
        , sum(case when Rev > 1000 then 1 else 0 end) as numitems1000
        , count(distinct CustID) as unique custs
        , count(distinct case when Rev > 1000 then custID else null end)
from
        SP
left join
        Sales
using( SalesID)
group by 1;
```

5. Write a query which returns IID and the average revenue generated for that item. Exclude items that have been sold less that 10 times.

```
select IID, avg( rev) as avgrev
from sales
group by 1
having count(1) > 10;
```

6. Items which are sold by salepeople in New York ("NY") have to add a tax of 5%. What is the total amount of tax that the company needs to add?

```
select
        .05 * sum( Rev ) as tax
from
        sales
where
        salesid in
                (select salesID from SP where st = 'NY')
```

7. Which item (IID) uses the most unique parts (PID)?

```
select
        IID
from
        Assembly
group by 1
order by count(1) desc
limit 1;
```

8. Which items uses the most *total* parts?

DRAFT

```
select
        IID
from
        Assembly
group by 1
order by sum(NoPart) desc
limit 1;
```

9. Return a list of the items (IID) that have never been sold:

```
select item.IID
from
        item
left join
        sales
using(IID)
where sales.IID is null
```

10. Return IID and the total cost of the parts used to make a one of them.

```
select
        IID, sum( price * noPart) as cost
from
        assembly
left join
        parts
using(PID)
group by IID;
```

11. Of all the salespeople from California ("CA") or Pennsylvania ("PA"), return the one (Name and SalesID) with the highest amount of revenue.

```
select
        SP.SalesID, Name
from
        SP
left join
        Sales
using( SalesID)
where SP.state in ('CA', 'PA')
group by 1,2
order by sum( Rev) desc
limit 1;
```

12. Of all the salespeople (SalesID) who have sold an item for more than \$100.00, return the average revenue per item sold on all of their sales.

```
select
        avg( rev) as ar
from
        sales
where
        salesid in
                (select distinct salesid from sales where rev > 100);
```

13. For each bonus structure ("H" or "L") return the percentage of revenue generated from salespeople in that bonus structure. This should return two rows and two columns.

```
select
        bonus,
        sr / sum( sr) over() as pct
from
        (select
                Bonus, sum(rev) as sr
        from
                SP
        left join
                sales
        using( SalesID)
        group by 1) as IQ
```

14. For each Bonus Structure ("H" or "L") return the percentage of revenue generated from salespeople in that bonus structure. This should return one row and two columns (PctRevH and PctRevL). Note that this is the same data as the previous query, but shaped wide, rather than long.

```
select
        sum( case when Bonus = 'H' then rev else 0 end )/sum( Rev) as PctRevH
        , sum( case when Bonus = 'L' then rev else 0 end )/sum( Rev) as PctRevL
from
        SP
left join
        sales
using( SalesID)
```

15. Calculate the total profit (total revenue - total cost) for each item (IID).

```
select
        (SR - ct * totalCost) as profit, lhs.IID
from
        (select sum(NoPart * Price ) as totalCost, IID
                from Assembly left join Part using(PID) group by 2) as lhs
left join
        (select sum( rev ) as SR, IID, count(1) as ct from sales group by 2) as lhs
using( IID )
group by 2;
```

16. Of all the items (IID) which use more than 5 unique PIDs, which two have the largest number of sales (in terms of count)?

```
select
        lhs.IID
from
        (select
                IID
        from
                Assembly
        group by 1
        having count(1) > 5 ) as lhs
left join
        Sales
using(IID)
group by 1
order by count(1) desc
limit 2;
```

17. An item is called “complex” if the ratio of unique parts (PIDs) to total parts is more than .90% and is called “simple” if the ratio is below .25%. Write a query which returns one row with three columns: the first column should be the total revenue generated by complex items, the second should be the total revenue generated by simple items and the third should be the revenue generated by items which are neither simple nor complex.

```
select
        sum( case when rat > .9 then amt else 0 end ) as TR_complext
        , sum( case when rat < .25 then amt else 0 end) as TR_Simple
        , sum( case when rat <=.9 and rat >= .25 then amt else 0 end) as TR_neither
from
        (select IID, count(1)::float / sum( NoPart) as rat from Assembly
        group by 1) as lhs
left join
        Sales
using(IID)
```

18. Return the long version of the query above. This time there will be two columns and three rows. An item is called “complex” if the ratio of unique parts (PIDs) to total parts is more than .90% and is called “simple” if the ratio is below .25%. The first column should be equal to “Complex”, “Simple” or “Neither” and the the second column should be the revenue generated by that type.

```
select
        case
                when rat > .9 then 'Complex'
                when rat < .25 then 'Simple'
                else 'Neither'
        end as typ
        , sum( amt )
from
        (select IID, count(1)::float / sum( NoPart) as rat from Assembly) as lhs
left join
        Sales
using(IID)
group by 1;
```

19. Which salesperson (SalesID) generated the highest profit (total revenue - total cost)?

```
select
        SalesID
from
        (select sum(NoPart * Price ) as totalCost, IID
                from Assembly left join Part using(PID) group by 2) as lhs
left join
        (select sum( rev ) as SR, IID, SalesID, count(1) as ct from sales group by 2,3) as lhs
using( IID )
group by 2
order by (SR - ct * totalCost) desc
limit 1;
```

20. The company believes that there is an assembly cost of roughly $1.00 per part. Note that if an item requires four of the same part, this means that the assembly cost is $4.00. Factoring in this cost, what item has the highest total profit?

```
select
        IID
from
        (select sum(NoPart * Price) + sum( NoPart)*1.0 as totalCost, IID
                from Assembly left join Part using(PID) group by 2) as lhs
left join
        (select sum( rev ) as SR, IID, count(1) as ct from sales group by 2) as lhs
using( IID )
group by 2
order by (SR - ct * totalCost) desc
limit 1;
```

21. For each item (IID), return the average number of days between sales (note that subtracting two dates yields the number of days between those dates).

```
select
        IID, avg(diff) as avgdiff
from
        (select
                IID
                ,dt -  lag(dt) over(partition by IID order by dt asc) as diff
        from
                sales ) as IQ
group by 1;
```

22. Using a cross join return a dataset which contains 3 columns: IID, month and the number of days that that IID was *not* sold that month. Assume that (a) the Sales table contains all information from 2011 and (b) that every possible sales date is represented in the Sales table. Note that if an item is not sold at all during the year it *should* still be in this result.

```
select
        IID, date_part('month', dt ) as mnth
        , sum( case when sales.dt is null then 1 else 0 end ) as daysmissing
from
        (select distinct IID from item) as lhs
cross join
        (select distinct dt from sales) as rhs1
left join
        sales
on lhs.IID = sales.IID and rhs1.dt = sales.dt
group by 1,2;
```

23. Write a query which returns 3 columns: date, the total revenue on that date and the average revenue from the last three days, but *not* including the current day. In other words if today is 1/5, then it should be the average revenue from 1/2, 1/3 and 1/4.

```
select
        dt, sr
        , avg( sr ) over (order by dt asc rows between 4 preceding and 1 preceding) as MA
from
        (select
                sum( rev) as SR, dt
        from
                sales
        group by 2 ) as iq
```

24. Which salesperson (name) sold the most number of *parts.* Note that this is not asking for number of *items* that each salesperson sold, but the number of parts contained within those items.

```
select
        Name
from
        SP
left join
        Sales
        using( salesid)
left join
        Assembly
        using( IID )
group by 1
order by sum(NoPart) desc
limit 1;
```

25. **Bonus** The salespeople in the table are characters from a movie. Which one?

DRAFT

6 2018 SQL Final

This exam was given at the masters level in 2018 for a course just covering SQL. Students had two hours to complete it.

The information below comes from an insurance company's database, which operates under the following model: Agents sell insurance to *households*. A household may purchase multiple types of insurance (e.g. fire, car, earthquake). If a household makes a claim on a policy, then the insurance company can either pay the claim or fight it. If the insurance company fights a claim, then there are expenses associated with it. This database contains only active policies.

- Columns with the same name can be assumed to be the same.
- **Unless stated above, all values are not null.**
- Agent Table:
  - This table contains information regarding the agents in the company.
  - **AgentID:** This the ID of the agent, it is unique to each row. (int)
  - **St:** This is the agent's state. NULL values mean that the agent is international. (string)
  - **Name:** This is the agent's name.
  - **Bonus:** This is equal to "H" or "L" for "High" or "Low" bonus structure. (string)
- Household table:
  - This table contains information about the households that are insured.
  - **HHID:** The unique ID for the household, it is unique to each row. (int)
  - **AgentID:** This is the ID of the agent who manages the household. (int)
  - **Name:** This is the head of household's name. (string)
  - **MultiFam:** Are there multiple people in the household (1 = "Yes", 0 = "No"). (int)
  - **zip:** The zip for the household. (string)
- Policy table:
  - This table contains information on each policy. A household may *multiple* policies.
  - **PolicyID:** The unique ID for each policy. You can assume it is unique to each row. (int)
  - **HHID:** The unique ID for the household. (int)
  - **PolicyType:** The type of insurance policy. (string)
  - **EndDate:** The date the policy expires. (date)
  - **Cost**: The cost charged for the policy. (float)
- Claims Table:
  - This table contains information on claims made against insurance policies. A policy *may* have multiple claims or *no* claims against it.
  - **ClaimID:** The ID associated with the claim, it is unique to each row. (int)
  - **PolicyID:** The policy the claim is against. (int)

DRAFT

- **Amt:** The amount of money being asked by the policy holder for the claim. (float)
- **dt:** This is the date that the claim was received. (date)

- Expense Table:
  - This table contains information on legal expenses for disputing a claim. A single claim may have multiple expenses.
  - **EID:** Expense ID, assume it is unique for each row. (int)
  - **ClaimID:** The claim that the expense is against. (int)
  - **EAmt:** The amount of the expense. (float)
  - **dt:** This is the ID of the salesperson. (int)
  - **type:** The type of expense (legal, private investigator, etc.). (float)

DRAFT

Table D.6: *Agent* Table, 228 Rows

AgentID	ST	Name	Bonus
1	CA	Miles Quaritch	H
2	PA	Jake Sully	L
3		Parker Selfridge	L
4	CA	Neytiri	H

Table D.7: *Household* Table, 4,525 Rows

HHID	AgentID	Name	Zip	MultiFam
1	1	John Smith	11217	1
2	1	James McWright	99924	1
3	37	Elrod Lee	12345	0

Table D.8: *Policy* Table, 5,587 Rows

PolicyID	HHID	PolicyType	EndDate	Cost
1	1	Fire	01-01-2019	1,245.76
2	1	Car	01-01-2019	2,247.05
3	5	Flood	07-08-2020	287.56
4	37	Earthquake	03-01-2019	22,476.18

Table D.9: *Claims* Table, 1,225 Rows

ClaimID	PolicyID	Amt	dt
1	22	254.85	09-03-2018
2	22	212.87	09-05-2018
3	188	12,285.96	07-07-2017

Table D.10: *Expense* Table, 2,258 Rows

EID	EAmt	ClaimID	dt	type
1	65.73	1	01-03-2011	Legal
2	75.73	12	02-03-2011	Private Invest.
3	265.04	17	01-05-2011	Filing Fees
4	554.36	8	08-27-2011	Court Fees
5	18.18	22	08-11-2011	Legal

DRAFT

For the following questions write a query that returns the answer **and ONLY the answer.** All questions should be answered with a single SQL query, unless stated otherwise. Do not use CTE's.

1. What are the top five agents (AgentID) in terms of *number* of Households sold to?

```
select AgentID
from Household
group by 1
order by sum(1) desc
limit 5;
```

2. How many Agent's have sold policy's to households in zipcode 11217?

```
select
        count(distinct agentID)
from
        Household
where zipcode = 11217;
```

3. Write a query which returns four columns. The first column should be the year (int), the second should be the month (int) that a policy expires, the third column should be the total costs of all policies that expire during that month-year and the final column should be the number of "Fire" policies that expire that month.

```
select
        date_part('year', EndDate) as yr
        , date_part('month', EndDate) as mnth
        , sum( Cost) as totalCost
        , sum( case when PolicyType = 'Fire' then 1else 0 end) as numberFire
from
        Policy
group by 1,2;
```

4. For each household (HHID), return (a) the HHID, (b) the number of policies it has (c) the total costs associated with those policies.

```
select
        hhid, count(1), sum(Cost)
from
        policy
group by 1;
```

5. Similar to the above question, return the (a) HHID, (b) the number of policies for that household and (c) the number of claims for that household.

DRAFT

```
select
        lhs.hhid, count(distinct rhs1.policyID), count(distinct rhs2.claimID)
from
        household as lhs
left join
        policy as rhs1
        using(hhid)
left join
        claims as rhs2
        using(policyid)
group by 1;
```

6. International Agents need to pay an additional tax on their policy's of 7.5% of the cost. For those policies which expire in 2019, what is the total international tax bill?

```
select
        sum(cost) *.075 as taxpaid
from
        agent
left join
        household
        using( agentID)
left join
        policy
        using( policyID)
where agent.st is null
and date_part('year', enddate) = 2019;
```

7. Write a query which returns the most common policy type (e.g. "Fire") for MultiFam households. Note that most common is the number of policies, NOT cost.

```
select
        policy.policytype
from
        household
        using( agentID)
left join
        policy
        using( policyID)
where multifam = 1
group by 1
order by count(1) desc. limit 1;
```

8. Write a query which returns two rows and two columns. The first column should be "Bonus" type ("H" or "L") and the second should be the number of agents who have that bonus type (number of rows).

```
select bonus, count(1)
from agent group by 1;
```

9. Write a query which returns one rows and two columns, which is the transpose of the previous

question. The first column should be "HighBonus" and contain the number of agents who have Bonus type of High and the second column should be the number of agents who have a Bonus type of Low ("LowBonus")

```
select
        sum( case when bonus = 'H' then 1 else 0 end) as HighBonus
        , sum( case when bonus = 'H' then 1 else 0 end) as LowBonus
from agent;
```

10. What is the total claim (*not policy cost*) amount by policy type?

```
select
        policytype
        , sum(amt) as claimamt
from
        policy
left join
        claims
        using(policyID)
group by 1
```

11. The company is interested in learning if policies from "H" bonus agents have more claims than those of "L" bonus agent. Please write query which returns the total policy costs as well as the total amount paid for claims, by Agent bonus type. This should return two rows and three columns (bonustype, policycosts, claimamt).

```
select
        bonus
        , sum( policy.cost) as policycost
        , sum( rhs2.amt) as claimamt
from
        agent
left join
        household
        using(AgentID)
left join
        policy
        using(HHID)
left join
        (select sum(amt) as amt, policyid from claims group by 2) as rhs2
        using(policyID)
group by 1;
```

12. Write a query which returns the following information: (a) AgentID, (b) The number of policies that they are in charge of, (c) the total number of claims against those policies and (d) the percentage of policies that the agent is in charge of that have a claim against it. Specifically, if a policy has two claims against it, then it should only be counted once in the numerator.

```
select
        AgentID
        , count(distinct policyID) as numPolicies
        , count(distinct claimID) as numclaims
        , count(distinct claimID)::float / count(distinct policyID) as pct
from
        household
left join
        policy
        using( HHID)
left join
        claims
        using( PolicyID)
group by 1;
```

13. Write a query which has three columns: year of policy expiration (as an integer), month of policy expiration (as an integer) and a running sum of the dollar amount of policies expiring over time. There should be one row for each year-month combination.

```
select yr, mon
        , sum( cost) over( order by yr asc, mon asc
                rows between unbounded preceding and current row)
from
(select
        date_trunc('year', enddate) as yr
        , date_trunc('month', enddate) as mon
        sum( cost) as cost
from
        policy
group by 1,2) as iq
```

14. We define a "house" risk zip code to be one where the total cost of "Fire", "Flood" and "Earthquake" is more than twice as large as the cost of "Car" policies within that zip code. What percentage of zip codes are high risk (return a single number, the percent of zip codes are "house" risk)? You may assume that there is at least one policy of each policy type within each zip code.

```
select
        avg( case when 2* hr > car then 1 else 0 end)  as pct
from
        (select
                zip
                , sum( case when PolicyType in ('Fire', 'Earthquake', 'Flood')
                        then cost else 0 end) as hr
                , sum( case when PolicyType = 'Car'
                        then cost else 0 end) as car
        from
                household
        left join
                policy
        using(policyID)
        group by 1) as innerq;
```

DRAFT

15. Write a query which returns the total expense amount of each “type” of Expense as well as the type of expense, making sure to sort from the largest total amount to the smallest total amount.

```
select
        type
        , sum(Eamt) as totalamt
from
        expenses
group by 1
order by 2 desc;
```

16. Write a query which returns the state with the largest number of “H” bonus agents. This should return 1 row and 1 column

```
select
        st
from
        agent
        where bonus = 'H'
group by 1
order by count(1) desc limit 1;
```

17. Of all the agents (agentID) who have ever sold a policy to zip code to 11217, which one had the highest average policy cost over all their policies that they sold?

```
select household.agentID
from
        household
left join
        policy
using( hhid)
where agentid in
        (select distinct agentid from household where zip = 11217)
group by agentID
order by avg( cost) desc limit 1;
```

18. Which household (HHID) has the most number of policies (assume that this is unique and that there exists a household with more policies than any other household)

```
select
        HHID
from
        policy
group by HHID
order by count(1) desc limit 1;
```

19. We are interested in the cash flow associated with claims, by policy type. Write a query which returns the 3 day moving average of the total claims (amt) received *per day* by policy type as well as the day-over-day percent change. There should be one row per day-policy type combination and four columns in the table (dt, policy type, moving average and percent change). To compute the percent

change, take today's total amount and divide by yesterday's. If today is 1/5/2018, then the three days in the moving average should be 1/3, 1/4 and 1/5. You can assume that every day of interest is represented in the policy table for all policies and that there are no days without claims for every policy type.

```
select
        dt, policytype
        , avg( amt) over( partition by policytype
                order by dt asc rows between 2 preceding and current row)
        , amt / lag( amt) over( partition by policytype
                order by dt asc )
from
(select
        sum( amt) as amt
        , dt
        , policytype
from
        policy
left join
        claims
        using( PolicyID)
        group by 2,3) as IQ
```

20. What month (just month, e.g. 12 for "December") is the most common end date for policies (by number of policies)?

```
select date_part('month', enddate)
from policy group by 1 order by count(1) desc limit 1;
```

21. It turns out that California Agents (those with state = "CA") were lying on costs associated with "Fire" Policies. In particular, every policy which was more than \$500 had an additional \$100 of fraud added to it. Write a query which returns two columns and one row. The first column should be the number of affected policies and the second should be the total amount (in dollars) of fraud.

```
select
        sum(1) as fclaims, 100 * sum(1) as dollarsfraud
from
        (select HHID, policyID, cost from policy
                where policytype = 'Fire' and cost > 1000) as lhs
left join
        household
        using(HHID)
where agentID in (select agentID from agent where st = 'CA') ;
```

22. What percentage of expenses have a claim amount associated with them larger than \$1000? Return a single number which is this percent.

```
select
        sum( case when claims.amt > 1000 then 1 else 0 end) / sum(1) as pct
from
        expense
left join
        claims
        using(claimID);
```

23. Write a query which returns three columns. The first column should be policy type, the second should be the number of Multifam households which purchased that policy type and the third should be the number of NON Multifam households (Multifam = 0) that purchased it. There should be one row per policy type.

```
select
        policytype
        , sum( case when multifam = 1 then 1 else 0 end)
        , sum( case when multifam = 0 then 1 else 0 end)
from household
left join policy using( HHID )
group by 1;
```

24. Which policy type has the highest average cost? Return one row and one column.

```
select policytype
from policy
group by 1
order by avg( cost) desc
limit 1;
```

25. Agent's are paid based on the following formula: If they have a "H" type bonus plan, then they receive \$100 for each policy they sell + 10% of the cost of that policy. If an agent has a low type ("L") bonus plan, they receive \$200 for each policy they sell + 5% of the cost of the policy. Compute the total wages paid to each agent. This should return two columns: agentID and their wages.

DRAFT

```
select
        agentID
        , case
                when bonus = 'H' then 100 * numpolicies + .1 * totalcost
                else 200 * numpolicies + .05 * totalcost
from
        (select
                agentID
                , max( bonus ) as bonus
                , count(1) as numpolicies
                , sum( cost) as totalcost
        from
                agent
                left join
                household
                using( agentID)
        left join
                policy
                using(HHID)
        group by 1) as innerQ
```

26. **Bonus**: All the agent's in the agent table are from a movie. What movie?

DRAFT

7 2019 Exams

In 2019, a joint Pandas/SQL five week course was taught which had four exams. These are all four exams. Note that each exam contained both Pandas and SQL questions.

Exam #1

The following table contains information about athletes at a college. You can assume each name uniquely defines a person and that a person only plays a single sport. There are no two rows with the same name.

- **name:** The name of the applicant (string)
- **wgt:** The athlete's weight (in kg) (float)
- **hgt:** The athlete's height (in meters) (float)
- **state:** The state that the athlete is from (string)
- **mdt:** The date that the measurement was taken (date type)
- **sport:** The sport (all lowercase) that the person plays (string)
- **sex:** Is the athlete male or female ("M" or "F") (string)
- **injury:** Injuries (all lowercase) are listed here (if Null that means no injury). There will be only a SINGLE injury listed (string)
- The name of the table / DataFrame is **ath**. No need to use a schema or load the DataFrame.
- Overly complex queries or code will be penalized.
- Only use syntax covered in class.

Figure D.3: *Ath* Table: 12,435 Rows

name	wgt	hgt	state	mdt	sport	sex	injury
Ringly Roberson	94.25	1.75	NY	8-1-2012	basketball	M	
Crash Bandicoot	88.25	1.62	MS	1-1-2012	rugby	M	shoulder
alligator reynolds	66.1	1.88	PA	8-5-2012	softball	F	

DRAFT

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Using SQL, write a query which returns the names (name only) of the top-6 tallest athletes who play basketball.

```
SELECT
        name
FROM
        ath
where sport = 'basketball'
order by hgt desc limit 6;
```

2. Using SQL, write a query which returns all basketball players (name only) shorter than 1.65 m from either New York ('NY') or Alabama ('AL'). Only include those athletes who are *not* injured.

```
select name from ath where
        state in ('NY', 'AL')
        and hgt < 1.65
        and sport = 'basketball'
        and injury is null;
```

3. Write an SQL query which returns the number of athletes from each state who *are* injured. This should be two columns by fifty rows.

```
SELECT
        state, count(1) as ct
from
        ath
where injury is not null
group by 1;
```

4. Write an SQL query which returns all rows and columns for female athletes.

```
select * from ath where sex = 'F';
```

5. We say that a sport is injury prone if 10% or more of the sport has injuries. Write an SQL query which returns a list of sports which are injury prone.

```
select sport from ath
group by 1
having sum( case when injury is not null then 1 else 0 end) ::float / sum(1) >= .10;
```

OR:

```
select sport from
(select sport
        , sum( case when injury is not null then 1 else 0 end)::float / sum(1) as rat
from ath
group by 1 ) as innerQ
where rat >= .1
```

6. We calculate the BMI ("Body Mass Index") of a person by taking their weight and dividing it by the height *squared*. Write a query which returns all rows and three columns: BMI, name and sport.

```
select wgt / hgt / hgt as bmi, name, sport
from ath;
```

7. Return a table with three columns: name, sport, and BMIFlag. BMIFlag should be equal to "0" if the BMI is less than or equal to 20, "1" if the BMI is greater than 20 and less than or equal to 30 and "2" otherwise.

DRAFT

```
select
        name, sport, wgt / hgt / hgt as BMI
        , case
                when wgt / hgt / hgt <= 20 then 0
                when wgt / hgt / hgt <= 30 then 1
                else 2 end as bmiflag
from
        ath;
```

8. If a person's BMI is greater than or equal to 30 they are defined as obese. Write a query which returns the *percentage* of athletes of each sport who are obese.

```
select
        sport,
        sum( case when wgt / hgt / hgt > 30 then 1 else 0 end )::float / count(1) as pctObese
from
        ath
group by 1;
```

9. Write a query which returns the sport, average height and average weight (by sport) for everyone whose name begins with the letter "A". Do not assume that the first letter of the person's name is always uppercase (there could be an "ann mitchell" in the table).

```
select
        sport
        , avg( hgt) as agh
        , avg(wgt) as agw
from ath
where upper(left(name,1)) = 'A'
group by 1;
```

10. Volleyball and basketball are known to be hard on the knees. Write a query which returns the percent of athletes who have "knee" injuries who play "volleyball" or "basketball" (combined) vs. the percent of knee injuries for sports which are NOT "volleyball" or "basketball". In other words, this should return two rows (one for volleyball / basketball and one for other) and two columns (one with a label for the sports included and one for the percent).

```
select
        case
                when sport = 'basketball' or sport = 'volleyball' then 'VB'
                else 'not VB'
        end as sportsType
, sum( case when injury = 'knee' then 1 else 0 end)::float / sum(1) as pctKnee
from ath
group by 1;
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that a DataFrame named *ath* is already loaded. Unless otherwise specified you may return either a Pandas Series or DataFrame.

DRAFT

1. Using Pandas, return an object (Series, DataFrame or array) of the names, and names only, of the top-6 tallest athletes who play basketball.

```
ath.loc[(ath.sport == 'basketball')].nlargest(6, hgt)['name']
```

2. Using Pandas, return all basketball players (name only) shorter than 1.65 m from either New York ('NY') or Alabama ('AL'). Only include those athletes who are not injured.

```
ath.loc[
(ath.sport=='basketball') & (ath.hgt < 1.65) & ((ath.state == 'AL') | (ath.state == 'NY'))
& (ath.injury.isna() == True), 'name']
```

3. Using Pandas, return an object which contains the names of sports (this should be without duplicates) which have a player with an injury to their "shoulder". You may assume that all the injuries in the table are lowercase.

```
ath.loc[(ath.injury=='shoulder')].sports.unique()
```

4. Female soccer players who have had a knee injury are going to be put on a special training program. Please return an object which contains *their names and only their names* sorted by state (A to Z).

```
ath.loc[ (ath.sport == 'soccer' ) & (ath.injury == 'knee') & (ath.sex == 'F')]
        .sort_values('state')['name']
```

5. We calculate the BMI ("Body Mass Index") of a person by taking their weight and dividing it by the height squared. In Pandas, return a DataFrame with three columns: BMI, name and sport for all rows.

```
ath = ath.assign(bmi = ath.wgt / ath.hgt / ath.hgt).loc[:, ['name', 'sport', 'bmi']]
```

6. Return a DataFrame with three columns: name, sport and BMIFlag. BMIFlag should be equal to "0" if the BMI is less than or equal to 20, "1" if the BMI is greater than 20 and less than or equal to 30 and "2" otherwise.

```
ath = ath.assign(bmi=ath.wgt / ath.hgt / ath.hgt).loc[:, ['name', 'sport', 'bmi']]
ath.loc[ (ath.bmi >= 30), 'bmiflag'] = 2
ath.loc[ (ath.bmi >= 20) & (ath.bmi < 30), 'bmiflag'] = 1
ath.loc[ (ath.bmi < 20), 'bmiflag'] = 0
```

7. There was an error with the weight machine and all weight-ins done in the month of March of 2012 were 10% too high. Please return an updated DataFrame with the information corrected. Note that this should include all rows and columns from the original dataset with wgt set 10% lower for miss-measured observations.

```
ath.loc[(ath.mdt.dt.year == 2012) & (ath.mdt.dt.month == 3), 'wgt']
        = .9* ath.loc[(ath.mdt.dt.year == 2012) & (ath.mdt.dt.month == 3), 'wgt']
```

8. Return a DataFrame which contains all information on anyone from Rhode Island ("RI") or who has a "knee" injury. Return this data sorted first by sport (A to Z), then by state (A to Z) and then by name (*Z to A*). Finally, upper case all returned names.

DRAFT

```
ath = ath.loc[(ath.state == "RI") | (ath.injury == "knee")]
        .assign(name=ath.name.str.upper())
        .sort_values(['sport', 'state', 'name'], ascending=[True, True, False])
```

9. Return a DataFrame which contains name, state and a flag which is equal to 1 if they play soccer and weight less than 70 kg or play basketball and weight less than 80 kg. The flag should be zero otherwise.

```
ath = ath.assign(flag = 0)
ath.loc[((ath.sport == "soccer") & (ath.wgt < 70))
        | ((ath.sport == "basketball") & (ath.wgt < 80)), 'flag'] =1
ath[['name', 'state', 'flag']]
```

Exam #2

The following table contains information about customer service interactions at a company. In particular, this has information about customers coming in and asking questions about their computer laptops.

- **serviceid:** This is an incrementing integer (int)
- **custid:** This is the ID for the customer (int)
- **pid:** This is the ID of the reported problem (e.g. battery problems are when pid = 2) (int)
- **servicedt:** This is the date that the service took place (date)
- **location:** This is the city and state of the service center (string)
- **result:** This is the diagnosis code for the device (e.g. result = 1 means solved) (int)
- **followup:** This contains information about if there was a follow up to the customer service (string)
- The name of the table / DataFrame is **cust**. No need to use a schema or load the DataFrame.
- The only column with Null values is "followup".
- Overly complex queries or code will be penalized.
- Only use syntax covered in class.

Figure D.4: *cust* Table: 12,435 Rows

serviceid	custid	pid	servicedt	location	result	followup
1	21	12	1-1-2012	Livermore, CA	1	
2	21	23	1-5-2014	Livermore, CA	1	
3	26	11	1-15-2018	Livermore, CA	110	Refund
4	53	18	3-3-2011	Yuma, AZ	1	

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Write a query which returns all rows and columns for services which occur in March or December of *any* year.

DRAFT

```
select
        *
from
        cust
where date_part('month', servicedt ) = 3
        or date_part('month', servicedt) = 12;
```

2. All customers with problem 22 (pid = 22) in California (location ends with 'CA') had the wrong result. Please write a query which returns a list of customers (no duplicates, just their IDs) who need to be notified.

```
select distinct custid
from cust
where right(location,2) = 'CA'
and pid = 22;
```

3. Write a query which returns a time series of the number of services which have *no* followup. This should be aggregated to the month/year level, so that all no-followup service of the same month/year are combined. Make sure to return the data sorted from earliest to latest date. In other words, there should be two columns: one indicating the year / month (as a date) and one with the number of services without a followup.

```
select
        date_trunc('month', servicedt) as monyear
        , sum(1) as ct
from
        cust
where followup is null
group by 1
order by 1 asc;
```

4. We say that a service request is solved if there is no followup *and* the result is equal to 1. For each problem type (pid), report the total number of solved service requests.

```
select
        pid
        , count(1) as solved
from
        cust
where followup is null and result = 1
group by 1;
```

5. We say that a service request is solved if there is no followup *and* the result is equal to 1. Generate a dataset which has one row per location and three columns. The first column is location, the second is the *total* number of solved interactions at that location (over all time) and the third is the total number of all customer service interactions in August of 2014 at that location.

DRAFT

```
select
        location
        , sum( case when result = 1 and followup is null then 1 else 0 end) as tst2
        , sum( case when date_trunc('month', servicedt)::date = '08-01-2014'
                then 1 else 0 end ) as tstaug
from
        cust
group by 1;
```

6. Write a query which returns the locations which have more than 10 different types of problems (unique pid). For those locations, return two columns: one with the original location and one *with just the state abbreviation*. You can assume that all locations are of the form "cityname, state abbreviation" and that all state abbreviations are TWO characters long.

```
select
        location, right(location, 2) as state
from
        cust
group by 1
having count(distinct pid) > 10
```

NOTE THE ABOVE CAN BE GROUP BY 1 OR GROUP BY 1,2

7. We are trying to figure out how effective each service center is at solving different problems. Create a dataset with 3 columns: the first should be location, the second should be problem (pid) and the third should be the *percent* of problems of that type and at that location which are "solved" (result = 1 and no followup). Make sure to exclude any problem/location group with less than or equal to 10 rows.

```
select
        pid
        , location
        , sum( case when result = 1 and followup is null
                then 1 else 0 end)::float / sum(1) as pct
from
        cust
group by 1,2
having count(1) > 10;
```

8. Write a query which returns the frequency distribution of different problem types. This should return two columns. The first, called num, should be the number of times a problem appears in the dataset and the second, called "val" should be number of times this frequency occurs. For example, let's say that problems (pid) 27, 35 and 115 each appear 8 times in the table (and no other problem appears exactly 8 times in the table), then there should be a row which is (8,3). Note that each problem (pid) should only be tallied once.

```
select ct as num, count(1) as val
from
        (select count(1) as ct
        from cust group by pid) as innerq
group by 1;
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that a DataFrame named *cust* is already loaded. Unless otherwise specified you may return either a Pandas Series or DataFrame.

1. Generate a DataFrame which contains all columns and only those rows which have problem #3 (pid = 3) and are solved (result = 1 and there is no followup).

```
cust.loc[ (cust.result == 1 ) & (cust.pid == 3) & (cust.followup.isna())]
```

2. Generate a dataset which contains (a) location (b) the earliest date that a service occurred for that location, (c) the latest date that a service occurred at that location, (d) the number of unique problem's (pid) that the location experienced and (d) the total number of services that occurred at that location. Don't worry about column names, but make sure that location is a *column*.

```
(cust
        .groupby(['location'])
        .agg( {'servicedt' : ['max', 'min'], 'pid' : ['nunique', 'sum']})
        .reset_index()
)
```

3. Generate a DataFrame with three columns: location, day of the week ("dow", as an integer) and the number of rows with *any, non-null* followup that were at that location on that day-of-the-week. Note that it doesn't matter if this returns columns or indexes for any value.

```
cust['dow'] = cust.servicedt.dt.dayofweek
cust['flag'] = 0
cust.loc[ ~(cust.followup.isna), 'flag'] = 1
cust.groupby(['location', 'dow']).agg({ 'flag' : 'sum'})
```

4. Generate a dataset which has one row per location and three columns. The first column is location, the second is the *total* number of solved (result = 1 and followup is empty) interactions at that location (over all time) and the third is the total number of customer service interactions in August of 2014 at that location. Make sure that this is three *columns*.

```
cust.assign(succ=0, Aug2014=0)
cust.loc[(cust.result == 1) & (cust.followup.isna()), 'succ'] =  1
cust.loc[(cust.servicedt.dt.month == 8) & (cust.servicedt.dt.year == 2014), 'Aug2014'] = 1
cust.groupby('location').agg({'succ' : ['sum'], 'Aug2014' : ['sum']}).reset_index()
```

5. We are trying to figure out how effective each service center is at each location at solving different problems. Create a DataFrame with three *columns*: the first should be location ("location"), the second should be problem ("pid") and the third ("succ") should be equal to 1 or 0, depending on if the outcome was solved (result = 1 and no followup) or not. This should have the same number of rows as the original DataFrame. Name this DataFrame "tst".

```
cust['succ'] = 0
cust.loc[(cust.result == 1) & (cust.followup.isna()), 'succ'] = 1
tst = cust.loc[ :, ['location', 'pid', 'succ']]
```

6. Assume that you have the "tst" DataFrame from the problem above. We now want to calculate

DRAFT

the percent of customer interactions which are solved, aggregated to the problem (pid) and location level. In other words, using the DataFrame from the previous problem, generate a new DataFrame consisting of three *columns*: location, pid and the percent of interactions which were solved. Specifically this is the sum of "succ" divided by the number of rows. Make sure to remove any row which has less than 10 observations in the original DataFrame (as there is not enough data to conclude anything from them).

```
tst = tst.groupby(['location', 'pid']).agg({'succ' : ['sum', 'count']})
tst.columns = ['s1', 'c1']
tst = tst.loc[ (tst.c1 > 10) ]
tst.assign(pct=tst.s1/tst.c1)[['pct']].reset_index()
```

Exam #3

The following table contains information about doctors and their patients. At most, each patient has one doctor.

- Columns in the patients table
  - **patientid:** This is an auto-incrementing integer for the patient (int)
  - **doctorid:** This is the ID for the doctor that they see (int)
  - **hgt:** This the height of the patient in meters (float)
  - **birthdt:** This is the date of birth of the patient (date)
  - **wgt:** This is the weight of the patient in kg (float)
  - **city:** This the city that the person lives in (string)
  - **state:** This is the state that the person lives in (string)
  - **sex:** The sex of the patient (M/F) (string)
- Columns in the doctors table
  - **doctorid:** This is an auto-incrementing integer for the doctor (int)
  - **speciality:** This is the type of doctor (string)
  - **surgeon:** Is the doctor a surgeon (Y/N) (string)
  - **sex:** The sex of the doctor (M/F) (string)
- The names of each table are "patients" and "doctors". No need to refer to any schema or load a DataFrame.
- **There are some patients who have not yet been assigned doctors, so doctorid could be Null in the patients table.**
- **There are some doctors who were just hired who have not been assigned patients yet.**
- Overly complex queries or code will be penalized.
- Only use syntax covered in class.
- Any two columns with the same name can be assumed to match.

DRAFT

Figure D.5: *Patients* Table: 12,435 Rows

patientid	doctorid	hgt	birthdt	wgt	city	state	sex
1	2	1.7	1-1-1997	58.2	Livermore	CA	M
2	18	1.65	1-5-1975	56.5	Livermore	CA	F
3	18	1.8	1-15-1994	67.3	Livermore	CA	M
4	7	1.93	3-3-1964	66.0	Yuma	AZ	M

Figure D.6: *Doctor* Table: 277 Rows

doctorid	speciality	surgeon	sex
1	Oncology	Y	M
2	ENT	N	F
3	General	N	M
4	Pediatric	Y	F

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Write a query which returns two columns. The first should be the doctorid and the second should be the number of patients that that doctor sees. This should be sorted from most to least patients seen. Make sure to *not* include doctors who have no patients and patients who have no doctors.

```
select
        doctorid, count(1) as ct
        patients
where doctorid is not null
group by 1
order by 2 desc;
```

2. A high-usage doctor is one that sees strictly more than 50 patients. Write a query which returns four columns. The first should be state, the second and third should be the average height and average weight of patients from that state and the fourth should be the number of patients from that state. Note that this should *only* include patients who have a high-usage doctor. There should be one row returned per state.

DRAFT

```
select
        state
        , avg( wgt) as aw
        , avg(hgt) as ah
        , count(1) as ct
from
        (select doctorid from patients
        group by doctorid
        having count(1) > 50) as lhs
left join
        patients
using(doctorid)
group by 1 ;
```

OR

```
select
        state
        , avg( wgt) as aw
        , avg(hgt) as ah
        , count(1) as ct
from
        patients
where doctorid in
        (select doctorid from patients
                group by doctorid
                having count(1) > 50)
group by 1 ;
```

3. Write a query which returns three columns. The first should be the speciality, the second should be the number of surgeons (surgeon = 'Y') within that speciality and the third should be the number of non-surgeons (surgeon = 'N') of that speciality.

```
select
        speciality
        , sum( case when surgeon = 'Y' then 1 else 0 end) as numsurg
        , sum( case when surgeon = 'N' then 1 else 0 end) as numnonsurg
from
        doctor
group by 1;
```

4. We say that a doctor is the same-sex as their patient if they are the same-sex as the patient (e.g. both Male or both Female). Write a query which returns two columns. The first should be the patientid and the second should be a flag which is equal to 1 if the patient and doctor are the same sex, zero otherwise. If a patient does not yet have a doctor the flag should be set to -1. This should have the same number of rows as the patients table.

```
select
        patientid
        , case
                when lhs.sex = rhs.sex then 1
                when rhs.sex is null then -1
                else 0
        end as flag
from
        patients as lhs
left join
        doctor as rhs
using(doctorid);
```

5. Create a dataset with five columns: year, speciality, number of patients who were born that year and have a doctor with that speciality, the average weight of patients who were born that year and have a doctor of that speciality and the average height of patients who were born that year and have a doctor of that speciality. Note that year should be returned as a number, not a date. Only include those patients who have been assigned doctors.

```
select
        date_part('year', birthdt) as yr
        , speciality
        , count(1) as numpatients
        , avg( hgt ) as avghgt
        , avg( wgt) as avgwgt
from
        patients
inner join
        doctor
using(doctorid)
group by 1,2;
```

6. We calculate the Body Mass Index of a person (BMI) as weight divided by height *squared*. Generate a table which has two columns: speciality and the max BMI of patients who see doctors of that speciality. Be careful to exclude doctors without patients and patients without doctors.

```
select
        speciality
        , max( wgt / hgt / hgt ) as maxbmi
from
        patients
inner join
        doctor
using(doctorid)
group by 1;
```

7. Write a query which returns four columns. The first should be speciality, the second should be sex (of the doctor), the third should be surgeon (the 'Y'/'N' flag) and the fourth should be the number of doctors of that type (where type is defined as speciality, sex and surgeon combination). If there is no doctor of that type then the count should be set to zero.

DRAFT

```
select lhs1.sex, lhs2.speciality, lhs3.surgeon, count(rhs.doctorid)
from
        (select distinct sex from doctor) as lhs1
cross join
        (select distinct speciality from doctor) as lhs2
cross join
        (select distinct surgeon from doctor) as lhs3
left join
        doctor as rhs
on lhs1.sex = rhs.sex
        and lhs2.speciality = rhs.speciality
        and lhs3.surgeon = rhs.surgeon
group by 1,2,3
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that a DataFrames named *patients* and *doctor* are already loaded. Unless otherwise specified you may return either a Pandas Series or DataFrame.

1. Generate a DataFrame with two columns. The first should be the doctorid and the second should be the number of patients that the doctor sees. This should be sorted from most to least patients seen. Make sure to *not* include doctors who have no patients and that it returns two *columns*.

```
(pd.merge(doctor, patients, on='doctorid', how='inner')
 .groupby( 'doctorid' )
 .agg({'patientid' : 'count'})
 .sort_values('patientid', ascending=False)
 .reset_index())

 This can be done without a merge, but you need to filter out the rows which don't match
```

2. Return a DataFrame which contains patientid, doctorid, birthdate and the speciality of the that patient's doctor. Only include those patients from California "CA" who have been assigned doctors.

```
lhs = patients.loc[ (patients.state == 'CA'), ['patientid', 'doctorid', 'birthdate'] ]
rhs = doctors.loc[ :, ['doctorid' , 'mtype']]
mrg = pd.merge( lhs, rhs, on='doctorid', how='inner')
```

3. A high-usage doctor is one that sees strictly more than 50 patients. Create a DataFrame which returns four *columns*. The first should be state, the second and third should be the average height and average weight of patients from that state and the fourth should be the number of patients from that state. Note that this should *only* include patients who have a high-usage doctor. There should be one row returned per state.

```
p1 = patients[['doctorid']].groupby('doctorid').agg({'doctorid' : ['count']}).reset_index()
p1.columns = ['doctorid', 'ct']
p1 = p1.loc[(p1.ct > 50)]

mrg = (pd.merge(p1, patients, on = ['doctorid'], how = 'inner')
        .groupby('state')
        .agg({'wgt' : ['mean'], 'hgt' : ['mean'], 'doctorid' : ['count']})
        .reset_index()
        )
```


DRAFT

OR

```
lst = patients[['doctorid']].groupby('doctorid').agg({'doctorid' : ['count']}).reset_index()
lst.columns = ['doctorid', 'ct']
lst = lst.loc[(lst.ct > 50), 'doctorid']

p1 = (patients.loc[(patients.doctorid.isin(lst)), :]
        .groupby('state')
        .agg({'wgt' : ['mean'], 'hgt' : ['mean'], 'doctorid' : ['count']})
        .reset_index()
        )
```

4. We calculate the Body Mass Index of a person (BMI) as weight divided by height *squared*. Generate a DataFrame which has two columns: speciality and the max BMI of patients who see doctors of that speciality. Be careful to exclude doctors without patients and patients without doctors. Make sure to return two *columns*

```
patients['bmi'] = patients.wgt / patients.hgt / patients.hgt

mrg = (pd.merge( patients, doctor, on='doctorid', how = 'inner')
        .groupby('speciality')
        .agg({'bmi' : ['max']})
        .reset_index()
       )
```

5. Create a DataFrame with three *columns*. The first should be state, the second should be number of patients from that state (total), the third should be the number of patients from that state which do NOT have doctors.

```
mrg = pd.merge( patients, doctor, on ='doctorid', how='left').assign(nodoc=0)
mrg.loc[ mrg.doctorid.isna(), 'nodoc'] = 1
mrg = mrg.groupby('state').agg({'doctorid' : ['count'], 'nodoc' : ['sum']}).reset_index()
```

6. How many patients do not have a doctor assigned?

```
patients.loc[patients.doctorid.isna(), 'patientid'].count()
```

Exam #4

The following tables contains information about Uber drivers, their rides and reviews.

- Columns in the *drivers* table:
  - **did:** This is an auto-incrementing integer ID for the driver (int)
  - **state:** This is the state that the driver lives in (string)
  - **prom:** Is the driver on a promotion? (Y/N) (string)
- Columns in the *rides* table:
  - **rid:** This is an auto-incrementing integer for the ride (int)
  - **ridets:** This is the date and time that the ride occurred (date)
  - **did:** This is ID for the driver (int)

DRAFT

- **air:** This is a flag (Y/N) for if the trip went to the airport (string)
- **length:** This is the ride length in km (float)

- Columns in the *reviews* table:
  - **rid:** This is the ride which was reviewed (int)
  - **review:** This is the review (star scale: 1 to 5) (int)
- The names of the tables and DataFrames are *drivers, rides* and *reviews.*
- Assume that there are no Null values in any of the tables.
- Overly complex queries or code will be penalized.
- Only use syntax covered in class.
- Do not create any views.
- Any two columns with the same name can be assumed to match.
- **Not all rides will have reviews. A ride can have, at most, one review.**
- **Not all Drivers may have rides. When a driver first signs up they will not have any rides.**
- **DO NOT USE CTE ("with"), but you CAN use any analytic / window function.**

did	state	prom
1	CA	Y
2	MN	N
3	CA	N
4	CA	Y

rid	ridets	did	air	length
1	1-1-2012 10:24 AM	45	N	1.25
2	12-23-2012 12:22 PM	45	N	23.45
3	7-6-2013 4:13 AM	112	Y	11.17
4	5-5-2014 1:23 PM	1125	N	.75

rid	review
1	5
23	4
35	4
45	1

Figure D.7: *driver* table (27,777 rows), *rides* table (454,123 rows) and *reviews* table (137,145)

SQL Section

Please answer the following questions making sure to return *only* the information requested.

1. Write a query which returns two columns and a row for each state in the table. The first column should be the state and the second should be the number of rides completed by drivers from that state.

```
select
        state
        , count(1)
from
        drivers
join
        rides
using( did )
group by 1;
```

DRAFT

2. Write a query which returns two columns and one row per driver. The first column should be the driver's ID number (did) and the second should be a Y/N flag if the driver's first ride was to the airport or not.

```
select
        distinct did, airflag
from
        (select
                did
                , first_value( air ) over(partition by did order by ridets asc) as airflag
        from
                rides ) as innerQ;

Note that you could use an aggregate function in the outer query, but there has to be an
inner query.
```

3. Write a query which returns the following: state of the driver, total rides completed by drivers from that state and the average review of drivers from that state. Make sure to sort this from highest to lowest average review. Exclude any state with strictly less than 1,000 riders served. This should have one row per state.

```
select
        state
        , sum(1) as totalrides
        , avg( review ) as avgreview
from
        drivers
join
        rides
        using( did )
left join
        reviews
        using( rid )
group by 1
having count(distinct rid ) >= 1000
order by 3 desc;
```

4. Write a query which returns three columns. The first column should be year (as an integer), the second column should be total rides from that year and the third should be the *running or cumulative total of all rides, excluding the current year*. There should be one row per year.

```
select
        yr
        , numrides
        , sum(numrides) over(order by yr asc rows between unbounded preceding and 1 preceding) as cumsum
from
        (select
                date_part('year', ridets) as yr
                , sum(1) as numrides
        from
                rides
        group by 1 ) as iq;
```

5. Write a query which returns four columns: The first should be the driver id ('did'), the second should be the total number of rides, the third should be the number of their rides with a review and the fourth should be the number of rides with 5-star reviews (review = 5). Only include rides from 2018

and make sure to sort the drivers from most to least rides with reviews. Exclude drivers without any rides.

```
select
        did
        , count(1) as numrides
        , sum( case when review is not null then 1 else 0 end) as num_with_reviews
        , sum( case when review = 5 then 1 else 0 end) as five_star_reviews
from
        rides
left join
        reviews
using(rid)
where date_part('year', ridets) = 2018
group by 1
order by 3 desc;
```

6. Write a query which returns two rows and two columns. One row contains the phrase "LT10" and have the average review for rides that were (strictly) less than 10 km and the other row should be "MT15" and should be the average review for rides which are (strictly) more than 15 km. There should only be two rows.

```
select
        case
                when length < 10 then 'LT10'
                when length > 15 then 'MT15'
        end as flag
        , avg( review)
from
        rides
join
        reviews
using(rid)
where length < 10 or length > 15
group by 1;
```

7. *Without* using an analytic function, return one row and two columns. This should be the transpose of the data in the previous question. The first column should be the average review for rides of less than (strictly) 10 km and the second should be the average review for rides of more than (strictly) 15 km. It should only have one row.

```
select
        avg( case
                when length < 10 then review
                else null
        end ) as lt10
        , avg( case
                when length > 15 then review
                else null
        end        ) as mt15
from
        rides
join
        reviews
using(rid);
```

Pandas Section

Please answer the following question, making sure to return only the information required. You can assume that DataFrames named *drivers*, *rides* and *reviews* are already loaded. Unless otherwise specified you may return either a Pandas Series or DataFrame.

1. Return a DataFrame which has two columns and a row for each state. The first column should be the state and the second should be the number of rides completed by drivers from that state.

```
pd.merge( rides, drivers, on='did', how='left')
        .groupby('state')
        .agg({'rid' : ['count']})
```

2. We are interested in studying the effect of driver promotion (prom = 'Y') on long rides (strictly greater than 10 km). Return a dataset which contains two rows (one for prom='Y' and one for prom = 'N') and has three *columns*. The first should be prom, the second should the number of long rides to the airport (air='Y') the third should be the number of long rides *not* to the airport (air='N').

```
mrg = pd.merge( drivers, rides, on='did', how='left')
mrg = mrg.loc[(mrg.loc[:, 'length'] > 10), :]

mrg.loc[ : , 'airflag' ]  = 0
mrg.loc[ (mrg.loc[:, 'air']) =='Y' , 'airflag' ]  = 1

mrg.loc[ : , 'Nairflag' ]  = 0
mrg.loc[ mrg.loc[:, 'air']=='N' , 'Nairflag' ]  = 1

mrg = (mrg
        .groupby('prom')
        .agg( { 'airflag' : ['sum'], 'Nairflag' : ['sum']})
        .reset_index()
        )
```

3. Return a DataFrame with two rows and two *columns*. One row contain the phrase "LT10" and then has the average review for rides that were (strictly) less than 10 km and the other row should be

"MT15" and should be the average review for rides which are (strictly) more than 15 km. There should only be two rows.

```
mrg = pd.merge( rides, reviews, on='rid', how='inner')

mrg = (mrg
        .loc[(mrg.loc[:, 'length'] < 10) | (mrg.loc[:, 'length'] > 15), :]
      )

mrg.loc[:, 'flag'] = 'LT10'
mrg.loc[(mrg.loc[:, 'length'] >15), 'flag'] = 'MT15'

mrg.groupby('flag').agg({'review' : ['mean']}).reset_index()
```

4. There is a worry that there is a relationship between ride length and the percentage of rides with reviews. To analyze this we will create a flag called "lflag", which is equal to 1 if the ride length < 2 km, 2 if the length is $>= 2$ and < 5 km and 3 if the length $>= 5$ km. Create a dataset which has the following columns: lflag, state, number of *rides* which are of that flag-state combination, the number of those rides with reviews and the average review from rides of that lflag-state combination.

```
mrg = pd.merge( drivers, rides, on ='did', how='inner')
mrg = pd.merge( mrg, reviews, on = 'rid', how='left')

mrg.loc[:, 'lflag'] = 1
mrg.loc[(mrg.loc[:, 'length']) >=2) & (mrg.loc[:, 'length'] <5) , 'lflag'] = 2
mrg.loc[(mrg.loc[:, 'length'] >5)  , 'lflag'] = 3

mrg.groupby( ['state', 'lflag'])
        .agg({ 'review' : ['mean', 'count'], 'rid' : ['count'] })
        .reset_index()
```

5. What is the average review for all rides from drivers which have *ever* had a ride over 60km? This should return a single number.

```
lst = rides.loc[(rides.loc[:, 'length'] > 60), 'did'].drop_duplicates()

rds = rides.loc[rides.loc[:, 'did'].isin(lst), :]

pd.merge(rds, reviews, on='rid', how='inner')['review'].mean()
```

8 USF's student table

Use the following tables when answering the questions. The tables below consistent of information from USF's undergraduate student system.

1. All columns with the same name (such as CID, SID, etc.) can be assumed to represent the same thing and join easily.
   - **SID** is an integer and is unique for each student.
   - **CID** is an integer and is unique for each class.
   - **PID** is an integer and is unique to each professor.
2. Student Table: Each row is a unique student currently enrolled at USF.
   - **BirthDate** is a date while **ParentCity** and **ParentState** represent the city and state where their parent's live while **CurrentCity** and **CurrentState** are where the student currently resides. Both are varchar
3. Classes Table: Maps currently enrolled students to what classes they enrolled in over the course of their studies.
   - **EnrollDate** is the date they they enrolled in the class.
   - **Semester** is the a varchar(10) which is equal to "Summer", "Spring" or "Fall." while **Yr** is the year, in integer form.
   - **Grade** is the grade they received. **If the student is currently taking the class or if the student withdraws the course, then grade is null**.
4. Catalog Table: Represents the list of classes available at USF for all currently enrolled undergraduates to take.
   - **Department**, **ClassName** and **Units** represent the department, name and number of units of each class.
5. WithDraw Table: Represents information about currently enrolled students who withdraw from a class.
   - **DropDate** is the date the student withdrew from the class.
6. Some other assumptions:
   - A student cannot repeat a class and a student can only withdraw from a class a single time. In other words, students get one shot at taking each class.
   - **All currently enrolled students are in the Class table.**

DRAFT

Figure D.8: *Student* Table, 4,525 Rows

SID	BirthDate	ParentCity	ParentState	CurrentCity	CurrentState	major
1	01-01-99	Oakland	CA	San Francisco	CA	Math
3	01-21-99	Fremont	CA	San Francisco	CA	Undeclared
2	11-08-98	Philadelphia	PA	Burlingame	CA	Comp. Sci.

Figure D.9: *Classes* Table, 72,485 Rows

SID	CID	EnrollDate	Semester	Yr	Grade
12	101	01-11-2015	Spring	2015	3.7
12	109	01-11-2015	Spring	2015	
12	800	01-10-2016	Spring	2016	0.0
12	1923	01-10-2016	Spring	2016	4.0
12	111	08-15-2016	Fall	2016	
12	546	08-15-2016	Fall	2016	
12	999	08-15-2016	Fall	2016	

Figure D.10: *Catalog* Table, 1,288 Rows

CID	PID	Department	ClassName	units
800	12	Math	Calculus I	4
801	22	Math	Calculus I	4
1118	102	English	Intro. to Shakespeare	2
45	888	Physics	Freshmen Seminar	2

Figure D.11: *Withdraw* Table, 14,888 Rows

SID	CID	DropDate
12	109	03-11-2015
1114	888	03-18-2015
765	2345	10-22-2015
9022	891	05-21-2015

DRAFT

1. Draw a picture showing the four tables and how they connect. Make sure to pay attention to which columns match with which table.
2. How many students are there currently enrolled?

```
select
        count(distinct sid)
from
        students;
```

3. What are the top 10 currently enrolled students (SID only) in terms of number of classes ever enrolled? E.g. which SID's enrolled in the most classes.

```
select
        sid
from
        classes
group by sid
order by count(1) desc
limit 10;
```

4. For each student (SID only), report the number of distinct departments that they have ever taken classes from.

```
select
        sid, count(distinct department) as numDepts
from
        students
left join
        catalog
using(cid)
group by 1;
```

5. What are the top five currently enrolled students (SID only) in terms of number of classes withdrawn?

```
select
        sid
from
        withdraw
group by sid
order by count(1) desc
limit 5;
```

6. Which department has the most popular major?

```
select
        major
from
        student
group by 1
order by count(1) desc
limit  1;
```

7. Which department has the largest number of currently enrolled students withdrawing from their classes?

```
select
        department
from
        withdraw
left join
        catalog
using(cid)
group by department
order by count(1) desc
limit 1;
```

8. For each student (SID only), report both how many classes they are enrolled in and how many they have withdrawn from.

```
select
        sid, count(classes.sid) as numEnrolled, count(withdraw.cid) as numWithDrawn
from
        classes
left join
        withdraw
using(sid, cid)
group by 1;
```

9. How many currently enrolled students have never withdrawn from a class?

```
select
        count( distinct sid)
from
        students
where sid not in
        (select distinct sid from withdraw);
```

We could also use a JOIN to answer this question:

```
select
        students.sid
from
        students
left join
        withdraw
on students.sid = withdraw.sid
group by 1
having count(withdraw.sid) = 0;
```

10. Which currently enrolled student (SID only) has the highest percentage of their classes withdrawn?

```
select
        sid
from
        classes
left join
        withdraw
using(sid, cid)
group by 1
order by count( withdraw.dropdate)::float / count( classes.sid) desc
limit 1;
```

11. Which department (department name) had the highest average time between date enrolled and date withdrawn (in number of days), for those currently enrolled students who withdrew from a class in that department?

```
select
        department
from
        classes left join withdraw using( sid, cid)
        left join catalog using( cid )
where dropdate is not null
group by 1
order by avg( dropdate - enrollate)  desc
limit 1;
```

12. Report, for each student the number of classes that they have taken in each department. Make sure to include rows (with a count of zero) for those departments from which a student has never taken a class.

```
select
        lhs.sid
        , rhs.department
        , count(classes.sid) as numclasses
from
        (select distinct sid from classes) as lhs
cross join
        (select distinct department from catalog) as rhs
left join
        classes
        on lhs.sid = classes.sid and rhs.sid = classes.sid
group by 1;
```

13. Of those currently enrolled students who withdraw from at least one class, what is the average number of classes that they withdrew from?

```
select avg( ct )
from
        (select count(1) as ct
        from withdraws group by sid)
        as innerQ;
```

14. Which professor (PID only) had the average highest percentage of their students withdraw from a class? This should be an average over classes as professors can teach multiple courses.

```
select pid, avg( pct) as apct
from (
        select pid, cid, sum(coalesce( wd, 0))::float / count(1) as pct
        from
                (select sid, cid from classes ) as lhs
        left join
                (select sid, cid, 1 as wd from withdraw) as rhs
                using(sid, cid)
        left join
                catalog
        using(cid)
        ) as innerQ
group by 1
order by 2 desc
limit 1;
```

15. Calculate each currently enrolled student's GPA, making sure to weigh it by the number of units.

```
select
        SID, sum( units * grade ) / sum( units) as GPA
from
        classes
left join
        catalog
using( cid )
where grade is not null
group by 1;
```

16. Of all classes with more than 15 currently enrolled students, which ones have an average grade given of more than 3.5? Make sure to not include withdraws.

```
select cid
from
classes
where grade is not null
group by 1
having count(1) > 15 and avg(grade) > 3.5;
```

17. Write a single query which calculates the average GPA of currently enrolled students who (1) withdraw from more than 10% of their classes and (2) withdraw from less than 10% of their classes.

```
select avg( GPA) as avgGPA, more_than10
from
(select
        sid
        , sum( units * grade ) / sum( units) as GPA
        , case when wd::float/ccount > .1 then 1 else 0 end as more_than10
from
        (select sid, sum( withdraw.dropdate) as wd, sum(1) as ccount
                from classes left join withdraw using( sid, cid)
                where grade is not null or dropdate is not null
                group by 1 ) as studentInfo
        left join classes using(sid)
        left join catalog using(cid)
        group by 1)
as innerQ
group by more_than10;
```

DRAFT

9 FF Sales Example

- In this section we are going to use the following set of hypothetical tables about a company (called F&F). When writing queries about this data you can assume that all tables are within the same schema.
- For this company, each transaction (or sale) has a single item and sales person attached to it.
- All columns with the same name can be assumed to match and merge.
- Transaction Table:
  - SID, ItemID and TID are all integers, Amount is a float and TransTS is a timestamp[1]
  - TID is unique per transaction and stands for "Transaction ID"
  - SID is unique per sales person and stands for "SalesPersonID"
  - ItemID is an ID that is unique to an item.
- Refund Table:
  - RefundTS is a timestamp
  - RefundAmount is a float, it is always less than or equal to the transaction amount
  - A transaction can only have a single refund, but not all transactions will have refunds.

Table D.11: *Transaction* Table, 12,525 Rows

SID	TransTS	ItemID	TID	Amount
1	01-01-14 08:10:25 PST	124	1	15000.18
3	01-21-14 18:10:25 PST	888	2	25000.45
2	11-08-14 12:09:25 PST	125	12	1854.65

Table D.12: *Refund* Table, 385 Rows

TID	RefundTS	RefundAmount
12	03-14-14 14:12:18 PST	1,854.65

Table D.13: *SalesPerson* Table, 50 Rows

SID	Name	MobilePhone	State	BonusStructure
1	Brian O'Conner	111-222-3333	CA	High
2	Dominic Torretto	444-555-6666	CA	High
3	Letty	777-888-9999	CA	High
4	Lightning McQueen	111-333-5555	AZ	Low
5	Tow Mater	222-444-6666	AZ	Low

DRAFT

[1] You can assume that the date functions introduced in class work on this data.

Table D.14: *Item* Table, 50 Rows

ItemID	BaseCost	Name
1	4.99	Washer Fluid
2	14.89	Brake Fluid
3	56.78	Brake Pads (Generic)

DRAFT

1. What are the top five sales people (SID only) in terms of number of sales?

```
select
    count(1) as numsales
    , sid
from
    transaction
group by 2
order by 1 desc
limit 5;
```

2. What are the top five sales people (Name) in terms of number of sales?

```
select
    count(1) as numsales
    , name
from
    transaction
left join
    salesperson
using( sid )
group by name
order by 1 desc
limit 5;
```

3. What are the top 10 sales people (Name) in terms of dollars of sales?

```
select
    name
from
    transaction
left join
    salesperson
using(sid)
group by 1
order by sum( amount) desc
limit 10;
```

4. Which mobile phone area code (first three digits) has the highest number of sales?

```
select
    left( mobilePhone, 3) as areaCode
from
    transaction
left join
    salesperson
    using(sid)
group by 1
order by sum( amount) desc
limit 10;
```

5. Calculate the total of revenue from each state.

```
select
    state
    , sum( amount) as state_amt
from
    transaction
left join
    salesperson
using(sid)
group by 1
```

6. Calculate the total revenue from all states.

```
select sum(amount) as totalsales from transaction;
```

7. Calculate the *percentage* of revenue from each state.

```
select lhs.state, lhs.state_amt / rhs.totalsales
from
    (select
        state
        , sum( amount) as state_amt
    from
        transaction
    left join
        salesperson
    using(sid)
    group by 1) as lhs
cross join
    (select sum( amount) as totalsales from transaction ) as rhs;
```

8. What was the total refunded amount for each sales person (SID only)?

```
select
    SID
    , sum( refundamount) as refamt
from
    transactions
left join
    refunds
using( TID )
group by 1;
```

9. How many sales people had no refunds? When thinking about this problem remember that a sales person has multiple transactions and each transaction *may* have a refund. We need to make sure that there are no refunds for any of the transactions for a sales person.

```
select
    sid
from
    transactions
left join
    refunds
using( TID )
group by 1
having count( refunds.refundamount ) = 0;
```

10. Which sales person (name only) had the highest percentage of refunds, based on number of transactions?

```
select
    sid
from
    transaction
left join
    refund
on transactions.tid = refund.tid
left join
    salesperson
on transaction.sid = salesperson.sid
group by 1
order by sum(refundamount)
```

11. For each salesperson (Name), what percentage of their sales were refunded?

```
select
    name
    , sum(refundamount)/sum(amount) as pct_refund
from
    transactions
left join
    salesperson
    using(sid)
left join
    refunds
    using(tid)
group by 1;
```

12. What is the average percentage refunded, on those transactions with refunds?

```
select
    avg( refundamount / amount ) as avg_ref_pct
from
    transactions
inner join
    refunds
    using( tid );
```

13. For each month, report the percentage of sales refunded by both number of refunds and dollars. Assume that a refund can occur in any month after a sale, but that all refunds are in these tables.

```
select
    date_part('month', transTS) as sales_month
    , count( refunds.refundamount )::float
        / count(transactions.amount) as pct_ref
    , sum( refunds.refundamount )
        / sum(transactions.amount) as pct_dol_ref
from
    transactions
left join
    refunds
using( tid )
group by 1;
```

14. What percentage of sales had (1) returns above 20% (by dollar) and (2) returns above 50% (by dollar) of their value? Write a single query that returns two values.

```
select
    sum( case when refundamount > .2 * amount then 1 else 0 end)
        / count(1) as pct_above_20
    , sum( case when refundamount > .5 * amount then 1 else 0 end)
        / count(1) as pct_above_20
FROM
    transactions
left join
    refunds
using(tid)
```

15. Let's calculate which item (Name) is the most returned, by percent of returns:

```
select
    item.name
from
    transactions
left join
    refunds
    using(tid)
left join
    item
    using( itemID)
group by item.name
order by count(refunds.tid)::float / count( transactions.tid) desc
limit 1;
```

16. Calculate the total amount of BaseCost returned, by item name.

```
select
    item.name
    , sum( BaseCost) as amtReturned
from
    refunds
left join
    transactions
    using(tid)
left join
    item
    using( itemID)
group by item.name;
```

DRAFT

10 The Sales Rollup

In this exercise we are going to write a ton of queries about an interesting sales dataset.

The Data

Table D.15: *Sales* Table, 10,250 Rows

SID	CID	amount	TID	sales_dt
1	4	13.55	1	1/11/2011
2	12	18.99	2	12/22/2012
2	12	22.01	3	1/12/2013

Table D.16: *Expenses* Table, 425 Rows

SID	CID	exp_dt	amount	e_type
12	18	5/4/2012	112.24	Dinner
2	44	10/10/2010	112.24	Sport

Table D.17: *Transactions* Table, 25,254 Rows

TID	IID	num
1	12	1
1	18	1
2	45	1
3	18	1

Table D.18: *Client* Table, 152 Rows

CID	Name	Address	City	st	Zip
1	John Smith	12 Blue Bell street	Hayward	CA	94552
2	Julia Xue	14 Howard Street	Danville	VA	24543

Table D.19: *SalesPerson* Table, 22 Rows

SID	Name	Address	City	st	Zip	StartDt	EndDt
1	Rachel Adams	27796 Hanover Hill	Hanover	NY	14081	12-12-2011	1-3-2014
2	Julia Xue	60 Darwin Court	Mobile	AL	36602	01-11-2012	7-7-2013

Table D.20: *Items* Table, 440 Rows

IID	ItemName	Cost
1	10 lbs. Concrete	12.95
2	20 lbs. Concrete	19.95
3	30 lbs. Concrete	27.95

Questions

Main Questions

1. How many sales people are there from CA?

DRAFT

Table D.21: *Regions* Table, 50 Rows

ST	Region
CA	West
PA	East
AL	South
VT	North

2. In how many different states does this company have clients?
3. What is the average, min and max cost of an item which is for sale?
4. What is the average, min and max cost of items which have concrete in their name?
5. What is the average, min and max cost of items which have concrete in their name vs. those which do not?
6. Write a query which returns the total sales and number of sales per month.
7. How many sales did each sales person have?
8. How many states are in each region?
9. How many transactions contained item 12?
10. What was the average number of items per transaction? (Think about what to do with the number of items column)
11. What are the top-5 items sold in units-sold?

A bit harder

1. The total dollars sold and number of items, per salesperson.
2. How many currently employed sales people?
3. Write a query which returns the total number of items that each salesperson sold.
4. What was the average amount expensed per-client and per-salesperson? (Need to do two queries?)
5. How many sales people are in each region?
6. How many sales are from each region? (How would you measure this?)
7. For each client, compute the Revenue - expenses.
8. For each client, compute the profit (revenue - expenses - costs).
9. For each client region, compute the total revenue.
10. For each salesperson region, compute the total revenue.
11. How many clients are in each region?
12. What is our profit per region? (How would you measure this?)
13. **How would you calculate profit per transaction?**
14. How many different sales people sold each item?
15. How many distinct items has each sales person sold?

DRAFT

16. For each region (based on client), return the # of salespeople servicing the region, the number of items sold in that region, the number of transactions that occurred in that region and the total cost of the items sold in that region.
17. Calculate, for each client region, the number of sales that occurred in the same region as the salesperson.
18. Because of weird tax rules, we need to collect a tax of 3% of revenue for those transactions where the client is in the western region. How much tax do we owe per month?

DRAFT

11 Sales Example I

This assignment contains information up to and including aggregate functions and dates. We will be using the table sp.mast which contains the following columns:

Column Name	Description
SID	This is the Salesperson's ID number.
spname	This is the Salesperson's name.
daysworked	The number of total days that the salesperson has worked.
itemssold	The total number of items that the salesperson sold.
bonus	If the salesperson was under the high- or low- bonus plan.
region	What region they worked in.
startdate	The date that they started.
salesdt	The date that the particular sales occurred.
descr	A description of the item sold.
cost	The cost of the item (in cents).
prc	The price of the item (in cents).

Figure D.12: Information regarding SP.MAST

1. How many total sales are in the database?

```
select sum(1) from sp.mast;
```

2. How many sales were completed each month?

```
select count(1), date_trunc('month', salesdt)
from sp.mast group by 2 order by 2;
```

3. How many sales were completed by region?

```
select count(1), region
from sp.mast group by 2 order by 2;
```

4. Using another tool (such as Excel or Google Docs) prepare a graph which contains the following information:

   (a) Month

   (b) Number of sales for that month

   (c) Total Revenue from sales that month

   (d) Total cost of items from that month

```
select
        count(1) as ct
        , date_trunc('month', salesdt)
        , sum(prc) / 100.0 as totalRev
        , sum(cost) / 100.0 as totalCost
from sp.mast group by 2 order by 2;
```

DRAFT

5. Using another tool (such as Excel or Google Docs) prepare a graph which shows, by region and month, the amount of *profit* generated. This should have four lines – one for each region.

```
select
        date_trunc('month', salesdt) as mnt
        , region
        , sum(prc - cost) / 100.0 as prft
from sp.mast
group by 1,2 order by 2,1;
```

6. Identify the top 7 sales people (name and SID) in terms of total revenue generated.

```
select spname, sid
from sp.mast
group by 1,2
order by sum( prc) desc limit 7;
```

7. Plot the monthly revenue (combined) for the top 7 salespeople.

```
select sum(prc) , date_trunc('month', salesdt) as mnt
from sp.mast
where sid in
        (select sid
        from sp.mast
        group by 1
        order by sum( prc) desc limit 7)
group by 2
order by 2;
```

8. Create a pie chart which breaks down all revenue into one of four categories: (a) the salesperson worked less than 10 days (b) the salesperson worked between 10 and 20 days (c) the salesperson worked between 20 and 50 days and (d) the salesperson worked more than 50 days.

```
select sum(prc) as rev,
        case
                when daysworked < 10 then 1
                when daysworked < 20 then 2
                when daysworked < 50 then 3
                else 4
        end
from
        sp.mast
group by 2;
```

9. Calculate the average profit *per region*. In particular, calculate the profit per region and then find the average over the regions.

```
select
        avg(prft) as prft
from
        (select sum(prc - cost) as prft
                from sp.mast
                group by region) as innerq;
```

10. We want to understand where we should concentrate our business – high margin items (which are those where the profit margin (price - cost)/cost >= 23%) or mid margin items (those where the profit margin is between 23% and 18%) or low-margin items (profit margin less than 18%). Create a dataset which identifies, for each distinct item, what margin group (high-, mid-, or low-) it is in.

```
select
        iid
        ,case
                when (prc-cost)::float / cost >= .23 then 'high'
                when (prc-cost)::float / cost >= .18 then 'mid'
                else 'low' end as margin
from
(select distinct prc, cost, iid from sp.mast) as innerQ
```

11. What was the average number of days that a salesperson worked? In particular, identify the number of days that each salesperson worked and then calculate the average of it.

```
select avg( days) as avgdays
from
        (select sid, max(daysworked) as days
        from sp.mast group by 1) as innerQ;
```

12. Salespeople are paid based on one of two plans: The “H” bonus plan which means that they are paid $130 per day, but receive a 10% commission or the “L” bonus plan which they are paid $150 per day, but receive a 5% commission. Calculate the amount of money that each salesperson made *on the non-commission* part.

```
select sid,
case when max(bonus) = 'H' then 130.0*max(daysworked)
else 150*max(daysworked) end as totalcomp
from sp.mast group by 1;
```

13. Calculate the total compensation paid to each salesperson, including both the commission and non-commission portion.

```
select
        sid
        , case
                when max(bonus) = 'H' then 130.0*max(daysworked) + sum(prc/100.0)*.1
                else 150*max(daysworked) + sum(prc/100.0)*.05 end as totalcomp
from sp.mast group by 1;
```

14. The company is thinking about changing the bonus plan so that the “H” bonus plan would be $100 per day, but 20% commission and the “L” bonus Play would be $175 per day with no commission.

DRAFT

Calculate the number of salespeople, within each bonus plant, that would be better off under the new vs. the old plan.

```
select count(1), bh, case when totalcomp > totalcomp2 then 1 else 0 end
from (
select
        sid, max(bonus) as bh
        , case
                when max(bonus) = 'H' then 130.0*max(daysworked) + sum(prc)/100.0*.1
                else 150*max(daysworked) + sum(prc)/100.0*.05 end as totalcomp
        , case
                when max(bonus) = 'H' then 100.0*max(daysworked) + sum(prc)/100.0*.2
                else 175*max(daysworked) end as totalcomp2

        from sp.mast

        group by 1) as iq group by 2,3;
```

15. Write a query which returns the following data per region:
    (a) The total profit (price - cost)
    (b) The average profit per salesperson
    (c) The total number of items sold

```
select
        region
        , sum(prc - cost) as prft
        , sum(prc - cost)::float / count(distinct sid) as pftPerPerson
        , count(1) as numSol
from
        sp.mast
group by 1;
```

16. Write a query which returns the following data, this time by margin – high margin items (which are those where the profit margin (price - cost)/cost >= 23%) or mid margin items (those where the profit margin is between 23% and 18%) or low-margin items (profit margin less than 18%).
    (a) The total profit (price - cost)
    (b) The average profit per item
    (c) The total number of items sold

DRAFT

```
select
        case
                when (prc-cost)::float / cost >= .23 then 'high'
                when (prc-cost)::float / cost >= .18 then 'mid'
                else 'low' end as margin

        , sum(prc - cost) as prft
        , sum(prc - cost)::float / count(1) as pftPeritem
        , count(1) as numSol
from
        sp.mast
group by 1;
```

DRAFT

12 Sales Example II

This assignment is a follow-up to the previous Sales Walk Through. This is the same company and data as the previous case, but now there are multiple tables, rather than a single combined table. The following is a data dictionary:

Column Name	Description
iid	This is the ID number of an item.
descr	A description of the item sold.
cost	The cost of the item (in cents).
prc	The price of the item (in cents).

Figure D.13: Information regarding sp.itemlist, which contains information about each item.

Column Name	Description
SID	This is the Salesperson's ID number.
iid	This is the ID number of an item.
salesdt	The date that the particular sales occurred.

Figure D.14: Information regarding sp.itemmap, which contains a map between salesperson, the date of the sale and what was sold.

Column Name	Description
SID	This is the Salesperson's ID number.
spname	This is the Salesperson's name.
daysworked	The number of total days that the salesperson has worked.
bonus	If the salesperson was under the high- or low- bonus plan.
region	What region they worked in.
startdate	The date that they started.

Figure D.15: Information regarding sp.sp, which contains information on each salesperson.

DRAFT

1. How many total sales are in the database?

```
select sum(1) from sp.itemmap;
```

2. How many sales were completed each month?

```
select count(1), date_trunc('month', salesdt)
from sp.itemmap group by 2 order by 2;
```

3. How many sales were completed by region?

```
select count(1), region
from
        sp.sp
left join
        sp.itemmap
using(SID)
group by 2 order by 2;
```

4. Using another tool (such as Excel or Google Docs) prepare a graph which contains the following information:
   (a) Month
   (b) Number of sales for that month
   (c) Total Revenue from sales that month
   (d) Total cost of items from that month

```
select
        count(1) as ct
        , date_trunc('month', salesdt)
        , sum(prc)/100.0 as totalRev
        , sum(cost)/100.0 as totalCost
from
        sp.itemmap
left join
        sp.itemlist
using(iid)
group by 2
order by 2;
```

5. Using another tool (such as Excel or Google Docs) prepare a graph which shows, by region and month, the amount of *profit* generated. This should have four lines – one for each region.

```
select
        date_trunc('month', salesdt) as mnt
        , region
        , sum(prc - cost)/ 100.0
from
        sp.itemmap
join
        sp.sp
using(sid)
join
        sp.itemlist
using(iid)
group by 1,2
order by 2,1;
```

OR:

```
select
    date_trunc('month', salesdt)::date as mnt
    ,sum (case when region = 'N' then prc-cost else 0 end ) as Npft
    ,sum (case when region = 'S' then prc-cost else 0 end ) as Spft
    ,sum (case when region = 'E' then prc-cost else 0 end ) as Epft
    ,sum (case when region = 'W' then prc-cost else 0 end ) as Wpft
from
    sp.itemmap
join
    sp.sp
    using(sid)
join
    sp.itemlist
    using(iid)
group by 1
order by 1;
```

6. Identify the top 7 sales people (name and SID) in terms of total revenue generated.

```
select
        spname, sid
from
        sp.itemlist
join
        sp.itemmap
using(iid)
join
        sp.sp
using(sid)
group by 1,2
order by sum( prc) desc limit 7;
```

7. Plot the monthly revenue (combined) for the top 7 salespeople.

DRAFT

```
select sum(prc)/100.0 , date_trunc('month', salesdt) as mnt
from
        sp.itemmap
join
        sp.itemlist
        using(iid)
where sid in
        (select
                sid
        from
                sp.itemlist
        join
                sp.itemmap
        using(iid)
        join
                sp.sp
        using(sid)
        group by 1
        order by sum( prc) desc limit 7)
group by 2
order by 2;
```

8. Create a pie chart which breaks down all revenue into one of four categories: (a) the salesperson worked less than 10 days (b) the salesperson worked between 10 and 20 days (c) the salesperson worked between 20 and 50 days and (d) the salesperson worked more than 50 days.

```
select sum(prc)::float/100 as rev,
        case
                when daysworked < 10 then 1
                when daysworked < 20 then 2
                when daysworked < 50 then 3
                else 4
        end
from
        sp.sp
join
        sp.itemmap
        using(sid)
join
        sp.itemlist
        using(iid)
group by 2;
```

OR

```
select
    sum( case when daysworked < 10 then prc else 0 end)/100.0 as C1
    ,sum( case when daysworked >= 10 and daysworked < 20 then prc else 0 end)/100.0 as C2
    ,sum( case when daysworked >= 20 and daysworked < 50 then prc else 0 end)/100. as C3
    ,sum( case when daysworked >= 50 then prc else 0 end)/100.0 as C4
from
    sp.sp
left join
    sp.itemmap
    using(sid)
left join
    sp.itemlist
    using(iid);
```

9. Calculate the average profit *per region*.

```
select
        avg(prft)/100.0 as prft
from
        (select sum(prc - cost)
                from
                        sp.sp
                join
                        sp.itemmap
                        using(sid)
                join
                        sp.itemlist
                        using(iid)

                group by region) as innerq;
```

10. We want to understand where we should concentrate our business – high margin items (which are those where the profit margin (price - cost)/cost >= 23%) or mid margin items (those where the profit margin is between 23% and 18%) and low-margin items (profit margin less than 18%). Create a dataset which identifies, for each distinct item, what margin group (high-, mid-, or low-) it is in.

```
select
        iid
        ,case
                when (prc-cost)::float / cost >= .23 then 'high'
                when (prc-cost)::float / cost >= .18 then 'mid'
                else 'low' end as margin
from
        sp.itemlist;
```

11. What was the average number of days that a salesperson worked?

```
select avg( daysworked)
from sp.sp;
```

12. Salespeople are paid based on one of two plans: The “H” bonus plan which means that they are paid \$130 per day, but receive a 10% commission or the “L” bonus plan which they are paid \$150 per

day, but receive a 5% commission. Calculate the amount of money that each salesperson made *on the non-commission* part.

```
select
        sid
        , case
                when bonus = 'H' then 130.0*daysworked
                        else 150.0*daysworked
                end as totalcomp
from sp.sp;
```

13. Calculate the total compensation paid to each salesperson, including both the commission and non-commission portion.

```
select
        sid
        , case
                when max(bonus) = 'H' then 130 *max(daysworked) + .1*sum(prc/100)
                else 150* max(daysworked) + .05*sum(prc/100)
        end as totalcomp
from
        sp.sp
left join
        sp.itemmap
        using(sid)
left join
        sp.itemlist
        using(iid)
group by 1;
```

14. The company is thinking about changing the bonus plan so that the "H" bonus plan would be $100 per day, but 20% commission and the "L" bonus plan would be $175 per day with no commission. Calculate the number of salespeople, within each bonus plant, that would be better off under the new vs. the old plan.

```
select count(1), bh, case when totalcompT <= totalcompN then 1 else 0 end as BetterOffFlag
from (
select
        sid, max(bonus) as bh
        , case
                when max(bonus) = 'H' then 130 *max(daysworked) + .1*sum(prc/100)
                else 150* max(daysworked) + .05*sum(prc/100)
        end as totalcompT
        , case
                when max(bonus) = 'L' then 130 *max(daysworked) + .1*sum(prc/100)
                else 150* max(daysworked) + .05*sum(prc/100)
        end as totalcompN
from
        sp.sp
left join
        sp.itemmap
        using(sid)
left join
        sp.itemlist
        using(iid)
group by 1) as iq
group by 2,3;
```

15. Write a query which returns the following data per region:
    (a) The total profit (price - cost)
    (b) The average profit per salesperson
    (c) The total number of items sold

```
select
        region
        , sum(prc - cost)/100 as prft
        , sum(prc - cost)::float / count(distinct sid)/100.0 as pftPerPerson
        , count(1) as numSol
from
        sp.sp
join
        sp.itemmap
        using(sid)
join
        sp.itemlist
        using(iid)
group by 1;

group by 1;
```

16. Write a query which returns the following data, this time by margin – high margin items (which are those where the profit margin (price - cost)/cost >= 23%) or mid margin items (those where the profit margin is between 23% and 18%) and low-margin items (profit margin less than 18%).
    (a) The total profit (price - cost)
    (b) The average profit per item
    (c) The total number of items sold

```
select
        case
                when (prc-cost)::float / cost >= .23 then 'high'
                when (prc-cost)::float / cost >= .18 then 'mid'
                else 'low' end as margin

        , sum(prc - cost)::float/100.0 as prft
        , sum(prc - cost)::float / count(1) / 100.0 as pftPeritem
        , count(1) as numSol
from
        sp.itemmap
join
        sp.itemlist
        using(iid)
group by 1;
```