Advanced Data Management for Data Analysis

Assignment 1

20-09-2020

Name

Titus Oosting

Student Number

s2683466

Introduction

Performing the TPC-H benchmark test with SF-1 and SF-3 using MonetDB and SQLite3 on a an ASUS VivoBook Flip TP412UA-EC053T

This report is structured as follows:

The full archive contains the following folders:

- `SQLite Queries` Adjusted Queries 01-22 (only for SQLite)
- `Output - MonetDB` Output files (.csv) for Queries 01-22 (SF-1) (MonetDB)
- `Output - SQLite` Output files (.csv) for Queries 01-22 (SF-1) (SQLite)
- `Adjusted CreateTables - SQLite` Adjusted 0-create_tables.sql for SQLite (includes constraints)

The following report contains additionally:

- Visualizations for the query sets for each SF, DBMS
- The query execution times achieved (with SF-1 & SF-3) e.g. `monet_timings_df`
- My own implementation of Q1 & Q6 (in Python) `q_01_python` & `q_06_python`

Configuration

Hardware

- 8 GB RAM
- Intel i5-8250U 1.6ghz
- 256GB SSD
- Windows 10 Home Edition 64bit

Software

- MonetDB v11.37.11

- SQLite v3.33.0
- DBeaver v7.2.0
- TPC-H v2.17.0
- Python v2.8
- pysqlite3 0.4.3
- python-monetdb 11.19.3.2

Parameters

- Using default configuration settings for SQLite and MonetDB

Installation, Setup & Queries

Installation

- Download [MonetDB Server & Client package]
- Create new folder 'ADMDB' in C:/Users/Titus/Appdata/Roaming/MonetDB5/dbfarm
- Find the 'M5server5.bat' file in C:/Program Files/MonetDB/MonetDB5, edit line 25&26: set path to above folder
- Download [SQLite bundle of tools]
- Download [DBeaver]

Setting up MonetDB - Client & Server

- Open MonetDB Server
- Open MonetDB Client
- Use existing user: 'monetdb' with identical password
- Run 0-create_tables.sql from MonetDB client `< C:\ADM\tpch_2_17_1\dbgen\MonetDB\0-create_tables.sql`
- Run 1-load_data.SF-1.sql from MonetDB client
- Run 2-add_constraints.sql from MonetDB client

Setting up DBeaver Connections

- Create a new Connection. Use 'monetdb' username and password. Use 'localhost' or '127.0.0.1' for field localhost.
- Create a new SQLite connection. Choose path to create database.

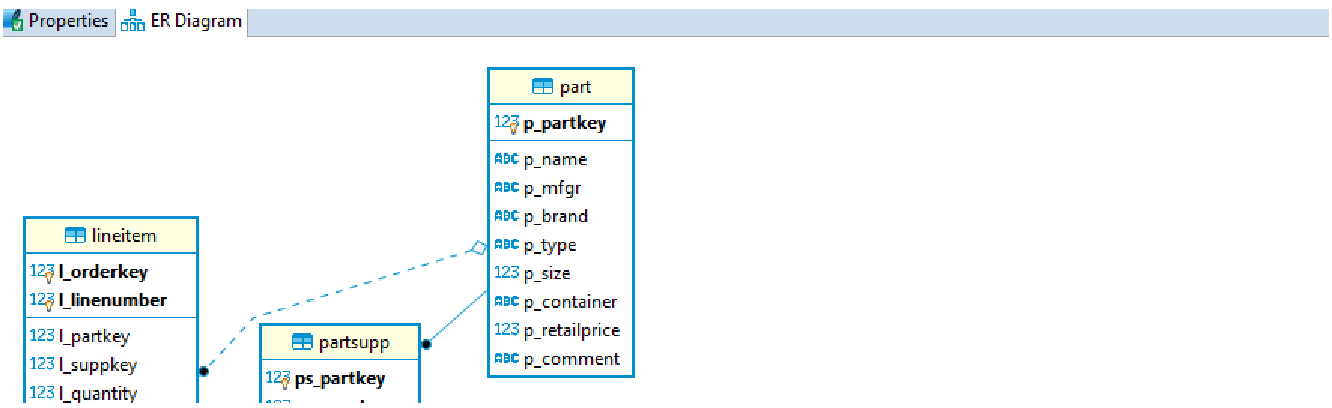
Setting up SQLite with DBeaver

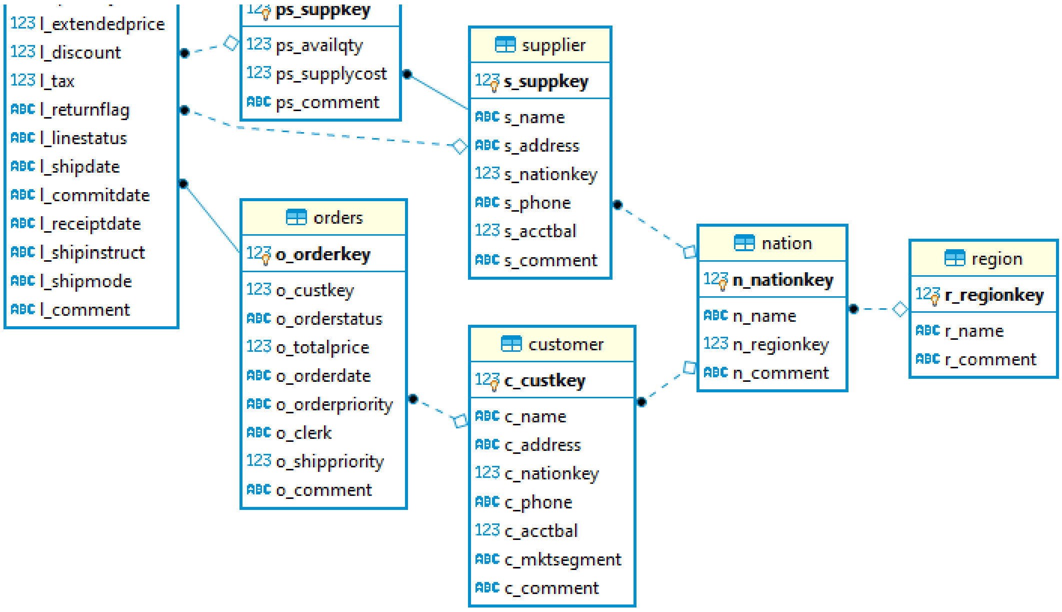
- Add constraints to the create tables script before running it in DBeaver
- For each table: copy table data into SQLite tables by rightclick import data - from equivalent MonetDB db
- Two tables were imported incorrectly. Lineitems and orders had wrong date format. Export these to .csv from MonetDB database, import .csv into SQLite database
- See below to verify correct constraints in SQLite tables

In [5]:

```
Image("C:/Users/Titus/Pictures/ADM/SQLiteER.png")
```

Out[5]:





Assignment Part One

Running Queries - MonetDB

- Run q01 through q22 through DBeaver. Export data to .csv on completion for verification.

Fixes

- **q14** - Too many digits (21) caused an error (18 max). Needed to cast: `then cast(l_extendedprice as double) * (1 - l_discount)`
- **q15** - Fix by running drop view separately after exporting through DBeaver

Running Queries - SQLite

- Run q01 through q22 through DBeaver. Export data to .csv on completion for verification.

Fixes

- **Multiple Queries** - fix date filters to conform to SQLite format. EG: `date '1998-12-01' - interval '90' day (3)` becomes `date('1998-12-01', '-90 days')`
- **q08** - Did NOT manage yet to fix precision error resulting in different results compared to benchmark
- **q13** - Updated syntax to work in SQLite3
- **q14** - Too many digits caused an error (18 max). Needed to add a cast: `then cast(l_extendedprice as double) * (1 - l_discount)`
- **q17** - Replaced inner-select with a 'temporary table' `q17-avg`
- **q20** - Updated syntax to work in SQLite3
- **q22** - Updated syntax and used views to work in SQLite3

Python Imports

In [1]:

```
import pandas as pd
from IPython.display import Image
from pandas.testing import assert_frame_equal
import timeit
from monetdb import mapi
import sqlite3
import datetime
import numpy as np
import seaborn as sns
```

In [2]:

```
from matplotlib import pyplot as plt
plt.style.use('ggplot')
```

Python Functions

In [3]:

```
query_names = [f"q{str(x).zfill(2)}" for x in range(1,23) ]
```

In [4]:

```
def check_query_results(query_names: list, dbms:str = "MonetDB"):
    """"Checks the results (read into Pandas Dataframe) against the benchmark."""
    for q in query_names:
        print(q)
        df1 = pd.read_csv(f"C:\ADM\Assignment1{dbms}Output\{q}.res.csv")
        df2 = pd.read_csv(f"C:\ADM\Assignment1Output\{q}.res.csv", header= None, names = list(df1.c
olumns))
        try:
            assert_frame_equal(df1, df2, check_dtype=False)
        except AssertionError as e:
            print("Results are different!!!")
            print(e)
    return
```

In [5]:

```
def monet_query_timings(query_names: list, server)-> dict:
    monet_timing_dict = dict((q,0) for q in query_names)
    for query in query_names:
        with open(f"C:/ADM/tpch_2_17_1/dbgen/MonetDB/{query}.sql") as f:
            x = 's' + f.read()
            func = %timeit -o -n 1 -r 20 server.cmd(x)
            monet_timing_dict[query] = func.timings
    return monet_timing_dict
```

In [6]:

```
def sqlite_query_timings(query_names: list, cursor)-> dict:
    sqlite_timing_dict = dict((q,0) for q in query_names)
    for query in query_names:
        with open(f"C:/ADM/tpch_2_17_1/dbgen/SQLite/{query}.sql") as f:
            x =  f.read()
            func = %timeit -o -n 1 -r 2 cursor.executescript(x)
            sqlite_timing_dict[query] = func.timings
    return sqlite_timing_dict
```

In [7]:

```
def q_01_python(lineitem_df: pd.DataFrame, q1_filter_date) -> pd.DataFrame:
    """Provided a Pandas DataFrame of Lineitem perform Query 1."""
    df = lineitem_df.copy()
    filtered_df =df[df.l_shipdate <= q1_filter_date]
    filtered_df['disc_price'] = filtered_df['l_extendedprice']* (1- filtered_df['l_discount'])
    filtered_df['charge'] = filtered_df['l_extendedprice']* (1- filtered_df['l_discount'])* (1 + fi
ltered_df['l_tax'])
    q1_result = filtered_df.groupby(['l_returnflag','l_linestatus']).agg({'l_quantity': ['sum', 'me
an', "count"],'l_extendedprice': ['sum', 'mean'],'disc_price': ["sum"], "charge":["sum"],
"l_discount": ["mean"] })
    return q1_result
```

In [8]:

```
def q_06_python(lineitem_df: pd.DataFrame, q6_filter_date)-> float:
    """Provided a Pandas DataFrame of Lineitem perform Query 6."""
```

```
    """Provided a Pandas DataFrame of Lineitem perform Query 6."""
    df = lineitem_df.copy()
    # use np.where to speed up
    q6_filtered_df = df[(df.l_shipdate >= q6_filter_date) &
                        (df.l_shipdate < q6_filter_date + datetime.timedelta(days = 365))&
                        (df.l_discount<= 0.07) & (df.l_discount>=0.05) & (df.l_quantity< 24)]
    q6_result =  np.sum(q6_filtered_df['l_extendedprice']* q6_filtered_df['l_discount'])
    return q6_result
```

MonetDB Query Verification

Below - Call a function (from collection above) to go through each of the output .csv and compare it to the provided results. The function checks for mismatches and prints the respective values in such case.

Note on q17 - Results from the provided .csv file have a different value than the .out file located in the dbgen/answers folder. Our value (same for SQLite) was consistent with the value in the .out file. See [this discussion] for more context (which conveniently indicates our output value was correct).

In [19]:

```
check_query_results(query_names = query_names)
```

```
q01
q02
q03
q04
q05
q06
q07
q08
q09
q10
q11
q12
q13
q14
q15
q16
q17
Results are different!!!
DataFrame.iloc[:, 0] are different

DataFrame.iloc[:, 0] values are different (100.0 %)
[left]:  [348406.054]
[right]: [3270416.82]
q18
q19
q20
q21
q22
```

SQLite Query Verification

Note on q08 - Values are close but technically incorrect. Precision error that I did not yet manage to resolve.

Note on q17 - refer to above

In [20]:

```
check_query_results(query_names = query_names, dbms = "SQLite")
```

```
q01
q02
q03
q04
q05
q06
q07
q08
```

```
Results are different!!!
DataFrame.iloc[:, 1] are different

DataFrame.iloc[:, 1] values are different (100.0 %)
[left]:  [0.03443589040665483, 0.041485521293530336]
[right]: [0.0344, 0.0414]
q09
q10
q11
q12
q13
q14
q15
q16
q17
Results are different!!!
DataFrame.iloc[:, 0] are different

DataFrame.iloc[:, 0] values are different (100.0 %)
[left]:  [348406.05428571376]
[right]: [3270416.82]
q18
q19
q20
q21
q22
```

MonetDB - Query Timing

Connect to MonetDB

In [50]:

```
server = mapi.Connection()
server.connect(username="monetdb", password="monetdb", hostname="localhost", database="ADMDB", port
=50000, language="sql")
```

SF-1 Timing

- Below functions record the time of 20 runs, 1 loop each, for each of the queries. The printed output is the mean and standard deviation.

In [51]:

```
result_dict = monet_query_timings(query_names, server)
```

```
258 ms ± 79 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 8.43 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
109 ms ± 99.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 7.71 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
82.8 ms ± 83.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 4.34 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
56.4 ms ± 29.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
71.6 ms ± 27.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
19.3 ms ± 7.42 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
129 ms ± 12.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
85.5 ms ± 26.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
123 ms ± 29.1 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
60.1 ms ± 16.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
36.9 ms ± 12.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
58.6 ms ± 8.65 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
134 ms ± 45.5 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 4.01 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
34.1 ms ± 15.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 4.56 times longer than the fastest. This could mean that an intermediate resu
```

```
lt is being cached.
54.6 ms ± 31.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
95 ms ± 8.65 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 6.33 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
105 ms ± 78.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
127 ms ± 15.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
75.3 ms ± 8.73 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
52.4 ms ± 15.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
182 ms ± 19.5 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
70.2 ms ± 6.64 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

- Get the data into nice formats for plotting

In [52]:

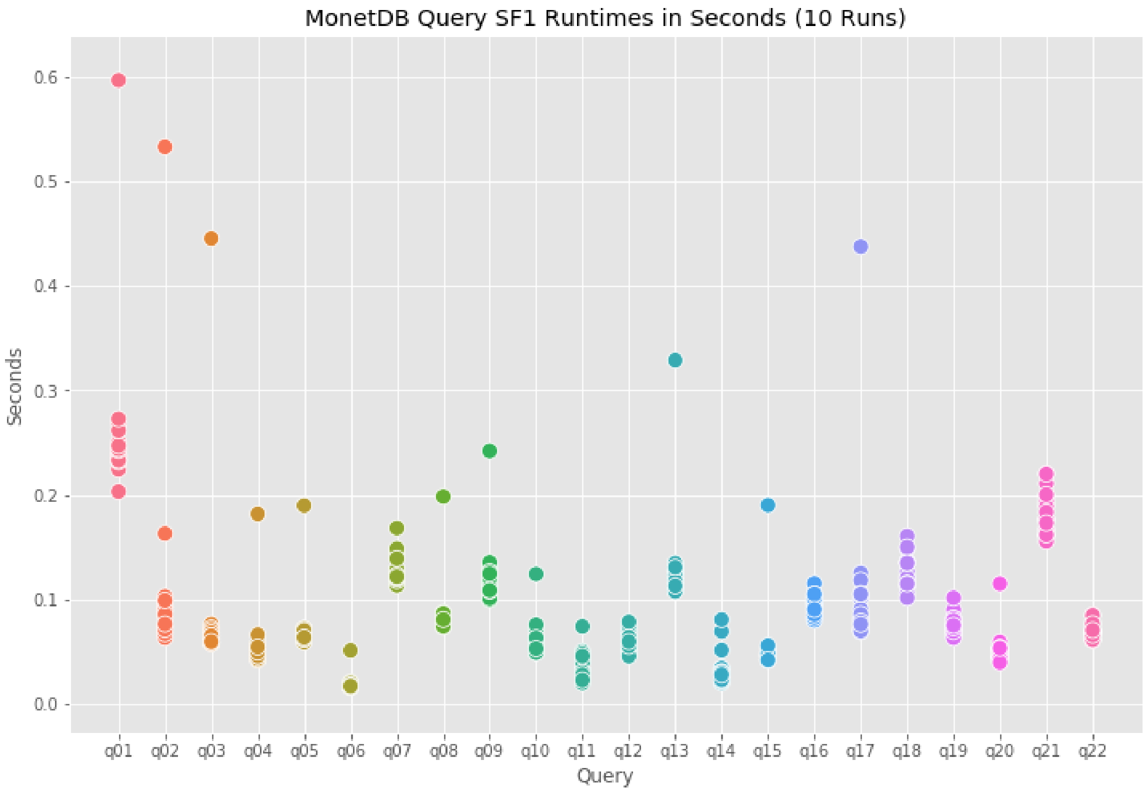
```
monet_timings_df = pd.DataFrame.from_dict(result_dict)
dfmm = monet_timings_df.melt(var_name='columns')
```

MonetDB Queries (SF1)

- For each of the queries, the 20 runtimes are mapped vertically (in seconds) in the graph below
- This gives an impression of the average runtime of each query, as well as outliers.
- Note on these outliers: eg for q01: the significantly higher run might have been the first before caching. As hinted in above function output: `This could mean that an intermediate result is being cached`

In [54]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = dfmm, hue= 'columns', x = 'columns', legend = None, s =100 )
plt.title("MonetDB Query SF1 Runtimes in Seconds (10 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```



- Below: data for the visualization

In [55]:

```
monet_timings_df
```

Out[55]:

	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	...	q13	q14	q
0	0.596796	0.533103	0.445349	0.181790	0.189782	0.051118	0.145239	0.198324	0.242088	0.124233	...	0.328828	0.034167	0.1901
1	0.203165	0.163020	0.076384	0.048581	0.067950	0.015622	0.113341	0.086574	0.099717	0.052278	...	0.131600	0.020119	0.0533
2	0.232926	0.102741	0.064206	0.043262	0.071407	0.016910	0.123987	0.077063	0.113680	0.061345	...	0.125622	0.028749	0.0470
3	0.234004	0.092846	0.062387	0.055046	0.069020	0.016165	0.168282	0.083715	0.107432	0.050973	...	0.115908	0.030980	0.0505
4	0.237654	0.097326	0.067474	0.041931	0.064204	0.019108	0.148601	0.080703	0.100914	0.050783	...	0.132141	0.030397	0.0525
5	0.224451	0.072778	0.061753	0.048712	0.060896	0.017366	0.117902	0.075814	0.112111	0.071535	...	0.109035	0.027786	0.0432
6	0.231790	0.068209	0.058783	0.057994	0.071917	0.016229	0.119861	0.079613	0.110441	0.048892	...	0.119392	0.032801	0.0458
7	0.245584	0.088388	0.060570	0.051481	0.072076	0.016491	0.125929	0.076912	0.129247	0.050329	...	0.127703	0.032036	0.0494
8	0.234300	0.063268	0.057737	0.044023	0.061422	0.017720	0.128848	0.084008	0.129835	0.052005	...	0.107645	0.034084	0.0457
9	0.233449	0.082183	0.059553	0.050220	0.061826	0.019152	0.126463	0.074653	0.121596	0.063236	...	0.123357	0.069345	0.0544
10	0.238150	0.093273	0.058344	0.048866	0.066957	0.018761	0.125983	0.087192	0.110283	0.062283	...	0.126860	0.024674	0.0436
11	0.250321	0.074750	0.057779	0.049392	0.071761	0.016790	0.126251	0.080497	0.108062	0.055334	...	0.133621	0.021474	0.0416
12	0.237053	0.067151	0.064380	0.043238	0.062657	0.019079	0.127950	0.079210	0.116974	0.055564	...	0.128264	0.022279	0.0466
13	0.233203	0.089984	0.059074	0.066225	0.070313	0.016387	0.120550	0.078473	0.108599	0.053219	...	0.122928	0.030681	0.0458
14	0.242670	0.074194	0.072633	0.051598	0.062710	0.020615	0.121221	0.075959	0.130991	0.075770	...	0.131635	0.030296	0.0462
15	0.268643	0.089397	0.070638	0.045869	0.060227	0.019748	0.123544	0.074946	0.135400	0.050698	...	0.113754	0.051367	0.0488
16	0.244037	0.085758	0.068156	0.050691	0.059168	0.017758	0.128685	0.073729	0.120441	0.049074	...	0.117613	0.029688	0.0479
17	0.247144	0.071208	0.065103	0.045909	0.061844	0.017252	0.123435	0.086578	0.126520	0.058290	...	0.134657	0.023133	0.0416
18	0.261622	0.098993	0.065282	0.049525	0.062317	0.016657	0.139061	0.074338	0.124589	0.063258	...	0.112866	0.080765	0.0558
19	0.272786	0.076500	0.059598	0.054464	0.063709	0.017057	0.121647	0.080801	0.108667	0.052884	...	0.130684	0.027903	0.0420

20 rows × 22 columns

SF-3 Timing

- The SF-3 data was loaded into a new schema in the MonetDB database. To run queries on that data set to that schema 'sf3'

In [57]:

```
server.cmd('sSET SCHEMA sf3;')
```

Out[57]:

```
'&3 0 0\n'
```

In [59]:

```
result_dict_sf3 = monet_query_timings(query_names, server)
```

```
786 ms ± 38.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
89.1 ms ± 33.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
208 ms ± 43.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
204 ms ± 17.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
206 ms ± 6.68 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
99.6 ms ± 7.22 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
368 ms ± 20.9 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
253 ms ± 7.84 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
386 ms ± 28.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
175 ms ± 11.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
48.5 ms ± 9.32 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 12.39 times longer than the fastest. This could mean that an intermediate res
ult is being cached.
277 ms ± 399 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
445 ms ± 144 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

```
110 ms ± 11.5 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
146 ms ± 58.1 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
233 ms ± 37.5 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 8.66 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
228 ms ± 248 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
308 ms ± 28 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
The slowest run took 14.00 times longer than the fastest. This could mean that an intermediate res
ult is being cached.
248 ms ± 401 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
166 ms ± 50.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
472 ms ± 32.7 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
179 ms ± 22 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

In [60]:

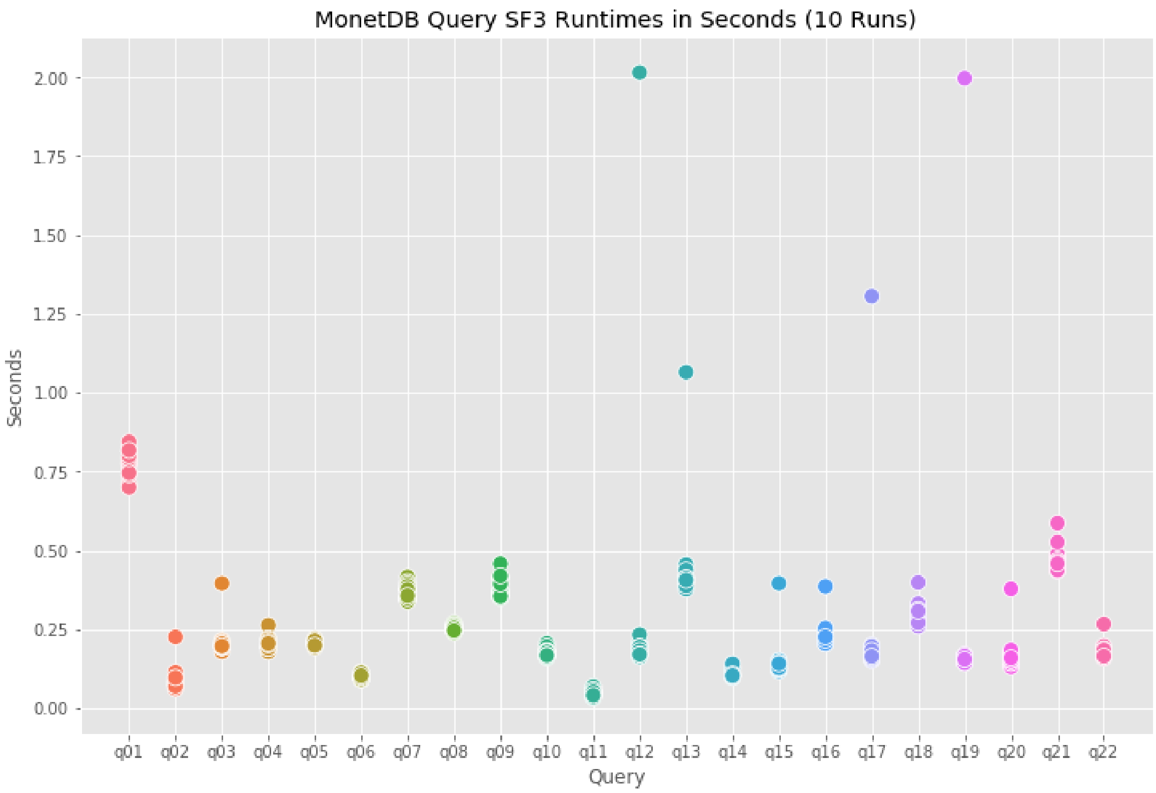
```
monet_timings_df3 = pd.DataFrame.from_dict(result_dict_sf3)
dfmm3 = monet_timings_df3.melt(var_name='columns')
```

MonetDB Queries (SF3)

- At a glance, query times are about 2.5 to 3.5 times that of the SF1
- The performance of queries relative to each other appears generally similar when comparing SF1 to SF3

In [61]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = dfmm3, hue= 'columns', x = 'columns', legend = None, s =100 )
plt.title("MonetDB Query SF3 Runtimes in Seconds (10 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```



In [62]:

```
monet_timings_df3
```

Out[62]:

q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	...	q13	q14	q

	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	...	q13	q14	q
0	0.709068	0.225524	0.395306	0.262649	0.210319	0.090802	0.354293	0.269384	0.375228	0.179462	...	1.065345	0.102648	0.3954
1	0.699834	0.075509	0.202187	0.219809	0.202902	0.114544	0.342426	0.261252	0.363293	0.205343	...	0.377337	0.105011	0.1167
2	0.739932	0.073395	0.178883	0.203485	0.205492	0.092278	0.377721	0.265390	0.365740	0.184735	...	0.453714	0.127056	0.1440
3	0.749923	0.093558	0.214551	0.178600	0.217894	0.090414	0.368221	0.253928	0.376554	0.167154	...	0.439647	0.110324	0.1506
4	0.815207	0.073954	0.203953	0.198257	0.204750	0.112318	0.372664	0.251210	0.361612	0.182128	...	0.403032	0.106501	0.1454
5	0.828622	0.075549	0.191751	0.215173	0.206910	0.099024	0.416111	0.242595	0.349694	0.195127	...	0.403802	0.129779	0.1238
6	0.815992	0.069239	0.192119	0.190106	0.204098	0.106904	0.374036	0.254793	0.372460	0.180228	...	0.416978	0.106024	0.1293
7	0.817497	0.102002	0.193571	0.214306	0.202377	0.098402	0.400317	0.262409	0.412750	0.167280	...	0.455018	0.100721	0.1505
8	0.777368	0.082115	0.196972	0.221433	0.211719	0.091137	0.374967	0.242710	0.409311	0.181540	...	0.414254	0.114337	0.1222
9	0.773415	0.113728	0.194698	0.191310	0.203473	0.100204	0.355487	0.251423	0.412615	0.174812	...	0.413452	0.110390	0.1319
10	0.776343	0.088858	0.192518	0.200784	0.192381	0.102883	0.345410	0.238971	0.391251	0.165438	...	0.397157	0.124292	0.1233
11	0.815302	0.086083	0.211209	0.194144	0.197845	0.099170	0.393456	0.252121	0.417822	0.179336	...	0.398062	0.140444	0.1248
12	0.815711	0.079651	0.198631	0.192324	0.207063	0.100953	0.382592	0.260555	0.409115	0.166814	...	0.438041	0.098778	0.1262
13	0.783860	0.075759	0.202737	0.216604	0.210727	0.112439	0.345347	0.251378	0.457713	0.163268	...	0.412870	0.097308	0.1259
14	0.745811	0.082455	0.205577	0.191498	0.205502	0.096949	0.345954	0.242814	0.366004	0.177747	...	0.408603	0.103093	0.1440
15	0.844371	0.084979	0.193246	0.193167	0.219291	0.092884	0.338395	0.253008	0.355566	0.168157	...	0.396986	0.100019	0.1245
16	0.794875	0.061408	0.204567	0.189594	0.199203	0.091275	0.386135	0.255024	0.354607	0.159715	...	0.398957	0.102303	0.1331
17	0.791358	0.071645	0.195945	0.200859	0.213170	0.099363	0.375844	0.249742	0.354646	0.162390	...	0.413019	0.108033	0.1373
18	0.800457	0.068996	0.204542	0.209776	0.214017	0.097692	0.348776	0.253900	0.393684	0.165797	...	0.387427	0.100807	0.1274
19	0.817741	0.096688	0.196488	0.204974	0.198693	0.103230	0.356517	0.245865	0.419419	0.167273	...	0.405849	0.103043	0.1409

20 rows × 22 columns

SQLite - Query Timing

Connect to SQL Lite

In [353]:

```
conn = sqlite3.connect('C:/SQLite/DBeaver/ADM')
cursor = conn.cursor()
```

SF-1 Timing

In [354]:

```
sqlite_timing_dict = sqlite_query_timings(query_names, cursor)
```

```
21.1 s ± 7.9 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
1.57 s ± 525 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
8.45 s ± 953 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
1.25 s ± 30.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
15.9 s ± 485 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
3.01 s ± 49.9 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
22.8 s ± 518 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
44.6 s ± 1.79 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
1min 47s ± 13.1 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
5.41 s ± 467 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
3.08 s ± 736 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
2.89 s ± 30.7 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
42 s ± 810 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
3.93 s ± 125 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
5.73 s ± 74.2 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
1.39 s ± 21.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
2min 30s ± 894 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
3.41 s ± 926 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
4.99 s ± 67.5 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
4.44 s ± 33.4 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
19.6 s ± 140 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

```
2.12 s ± 36.3 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

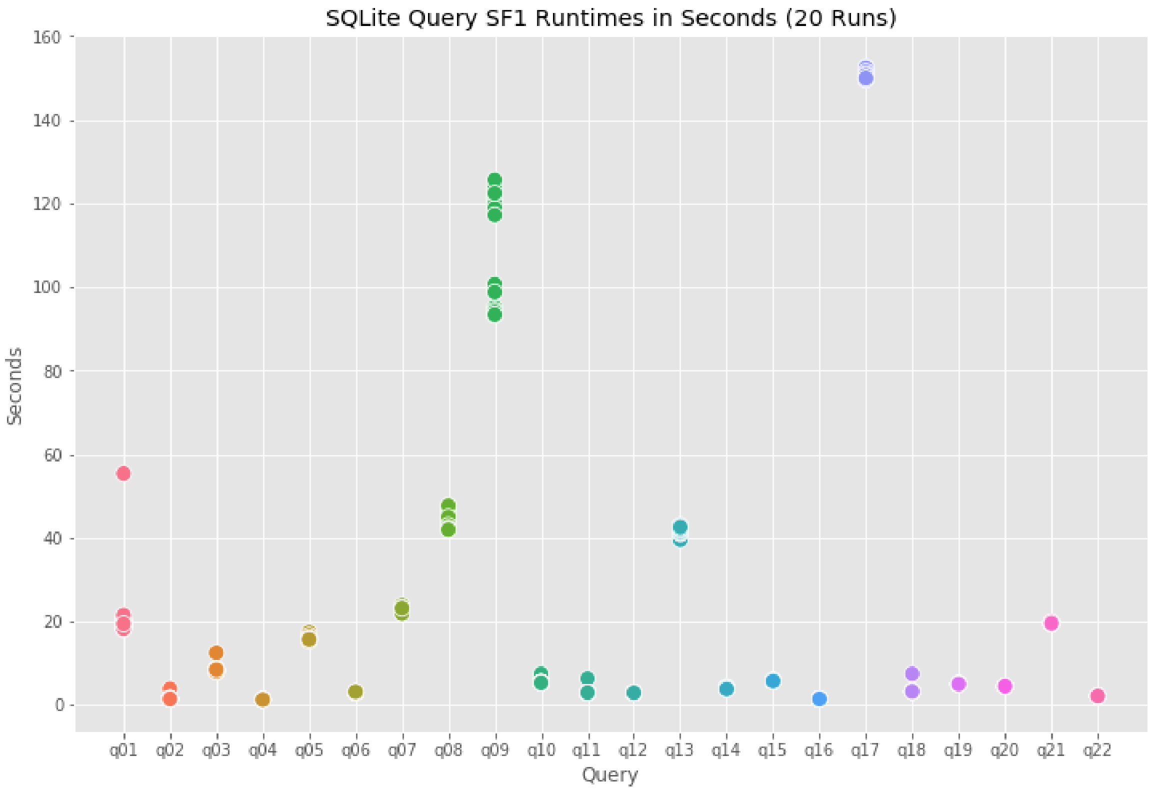
In [355]:

```
sqlite_timings_df = pd.DataFrame.from_dict(sqlite_timing_dict)
dfm = sqlite_timings_df.melt(var_name='columns')
```

All SQLite Queries (SF1)

In [370]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = dfm, hue= 'columns', x = 'columns', legend = None, s =100 )
plt.title("SQLite Query SF1 Runtimes in Seconds (20 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```



SF-3 Timing

- The SF3 data was loaded into a new database, so a new connection is made below.
- For the SF3 timing, real-life time constraints dictated that each query only be run twice (instead of the above 20 runs)

In [9]:

```
conn = sqlite3.connect('C:/SQLite/DBeaver/ADMSQLiteDBSF3')
cursor = conn.cursor()
```

In [10]:

```
sqlite_timing_dict3 = sqlite_query_timings(query_names, cursor)
```

```
1min 47s ± 1.57 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
17.7 s ± 8.97 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
1min 45s ± 57.2 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
7.32 s ± 31.4 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
1min 57s ± 55 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
17.1 s ± 250 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
```

```
2min 58s ± 2.25 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
4min 38s ± 5.2 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
10min 6s ± 49.9 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
15.8 s ± 293 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
14.8 s ± 5.28 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
8.15 s ± 48.1 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
2min 13s ± 1.94 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
14.6 s ± 3.68 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
16.4 s ± 197 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
4.53 s ± 361 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
10min 26s ± 40.6 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
13.7 s ± 4.24 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
15.4 s ± 218 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
15.4 s ± 1.86 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
2min 9s ± 9.72 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
7.81 s ± 739 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)
```

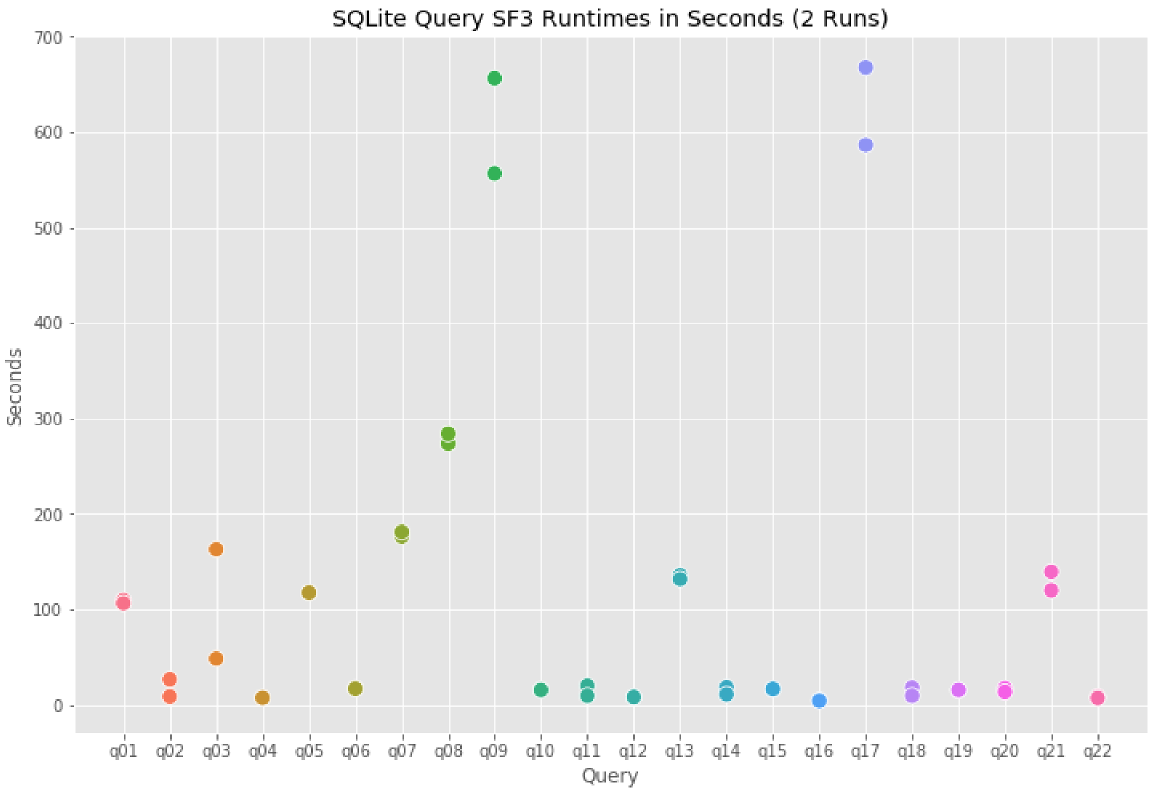
In [11]:

```
sqlite_timings_df3 = pd.DataFrame.from_dict(sqlite_timing_dict3)
dfm3 = sqlite_timings_df3.melt(var_name='columns')
```

SQLite Queries (SF3)

In [14]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = dfm3, hue= 'columns', x = 'columns', legend = None, s =100 )
plt.title("SQLite Query SF3 Runtimes in Seconds (2 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```



In [13]:

```
sqlite_timings_df3
```

Out[13]:

	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	...	q13

	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	...	q13
0	109.567640	26.622388	162.822908	7.288262	117.654178	17.391352	176.430559	273.559647	656.19311	16.103180	...	135.598934
1	106.422741	8.690580	48.360160	7.351119	117.544154	16.890460	180.921946	283.959798	556.42914	15.517261	...	131.712193

2 rows × 22 columns

Assignment Part Two

Python Query 1 Verification

- Read in SF1 Data

In [21]:

```
lineitem_df = pd.read_csv("C:/ADM/lineitem_202009191534.csv")
```

In [67]:

```
lineitem_df3 = pd.read_csv("C:/ADM/Lineitem Export SF3.csv")
```

In [22]:

```
lineitem_df.shape
```

Out[22]:

```
(6001215, 16)
```

In [68]:

```
lineitem_df3.shape
```

Out[68]:

```
(17996609, 16)
```

In [23]:

```
lineitem_df.head()
```

Out[23]:

	l_orderkey	l_partkey	l_suppkey	l_linenumber	l_quantity	l_extendedprice	l_discount	l_tax	l_returnflag	l_linestatus	l_shipdate	l
0	1	155190	7706	1	17.0	21168.23	0.04	0.02	N	O	1996-03-13	
1	1	67310	7311	2	36.0	45983.16	0.09	0.06	N	O	1996-04-12	
2	1	63700	3701	3	8.0	13309.60	0.10	0.02	N	O	1996-01-29	
3	1	2132	4633	4	28.0	28955.64	0.09	0.06	N	O	1996-04-21	
4	1	24027	1534	5	24.0	22824.48	0.10	0.04	N	O	1996-03-30	

In [26]:

```
lineitem_df['l_shipdate'] = lineitem_df['l_shipdate'].apply(lambda x: datetime.datetime.strptime(x,
'%Y-%m-%d'))
```

In [69]:

```
lineitem_df3['l_shipdate'] = lineitem_df3['l_shipdate'].apply(lambda x: datetime.datetime.strptime(
x, '%Y-%m-%d'))
```

Query 1

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date('1998-12-01', '-90 days')
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
 `
```

In [27]:

```
q1_filter_date = datetime.datetime.strptime('1998-12-01', '%Y-%m-%d') - datetime.timedelta(days = 9
0)
```

In [ ]:

```
q1_result = q_01_python(lineitem_df, q1_filter_date)
```

- Some formatting for comparison with benchmark results

In [29]:

```
q1_result.reset_index(inplace=True)
```

In [30]:

```
q_1_result_columns = ['l_returnflag',
'l_linestatus','sum_qty','avg_qty','count_order','sum_base_price','avg_price',
                      'sum_disc_price','sum_charge','avg_disc']
q_1_columns = ['l_returnflag',
 'l_linestatus',
 'sum_qty',
 'sum_base_price',
 'sum_disc_price',
 'sum_charge',
 'avg_qty',
 'avg_price',
 'avg_disc',
 'count_order']
q1_result.columns = q_1_result_columns
```

In [31]:

```
# Python Result
q1_result[q_1_columns]
```

Out[31]:

	l_returnflag	l_linestatus	sum_qty	sum_base_price	sum_disc_price	sum_charge	avg_qty	avg_price	avg_disc	count_ord
0	A	F	37734107.0	5.658655e+10	5.375826e+10	5.590907e+10	25.522006	38273.129735	0.049985	147849
1	N	F	991417.0	1.487505e+09	1.413082e+09	1.469649e+09	25.516472	38284.467761	0.050093	3885
2	N	O	74476040.0	1.117017e+11	1.061182e+11	1.103670e+11	25.502227	38249.117989	0.049997	292037
3	R	F	37719753.0	5.656804e+10	5.374129e+10	5.588962e+10	25.505794	38250.854626	0.050009	147887

In [32]:

```
# TPC-H Result
q1_df = pd.read_csv(f"C:\ADM\Assignment1Output\q01.res.csv",header = None, names = q_1_columns )
q1_df
```

Out[32]:

	l_returnflag	l_linestatus	sum_qty	sum_base_price	sum_disc_price	sum_charge	avg_qty	avg_price	avg_disc	count_ord
0	A	F	37734107.0	5.658655e+10	5.375826e+10	5.590907e+10	25.522006	38273.129735	0.049985	147849
1	N	F	991417.0	1.487505e+09	1.413082e+09	1.469649e+09	25.516472	38284.467761	0.050093	3885
2	N	O	74476040.0	1.117017e+11	1.061182e+11	1.103670e+11	25.502227	38249.117989	0.049997	292037
3	R	F	37719753.0	5.656804e+10	5.374129e+10	5.588962e+10	25.505794	38250.854626	0.050009	147887

In [34]:

```
# they are equal!
assert_frame_equal(q1_df, q1_result[q_1_columns], check_dtype=True)
```

Python Query 1 Timings

SF1

In [40]:

```
func1 = %timeit -o -n 1 -r 20 q_01_python(lineitem_df, q1_filter_date)
func1.timings
```

```
C:\Users\Titus\anaconda3\lib\site-packages\ipykernel_launcher.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
C:\Users\Titus\anaconda3\lib\site-packages\ipykernel_launcher.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
3.08 s ± 142 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

Out[40]:

```
[2.9958811000001333,
 3.0305323000000044,
 3.256669699999975,
 3.1752318999999716,
 3.024105200000122,
 2.957718300000124,
 2.971613599999955,
```

```
2.9215979999999035,
 2.9258734000000004,
 2.947681399999965,
 2.9487775999998576,
 2.9853928999998516,
 3.1378764000000956,
 3.429341400000112,
 3.271930399999974,
 3.1198452999999517,
 3.0371334999999817,
 3.0366014000001087,
 3.181288699999868,
 3.3089406999999937]
```

SF3

In [70]:

```
func1_sf3 = %timeit -o -n 1 -r 20 q_01_python(lineitem_df3, q1_filter_date)
func1_sf3.timings
```

```
C:\Users\Titus\anaconda3\lib\site-packages\ipykernel_launcher.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
C:\Users\Titus\anaconda3\lib\site-packages\ipykernel_launcher.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
1min 29s ± 29.3 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

Out[70]:

```
[66.25782590000017,
 90.18191630000001,
 67.76949259999992,
 93.78741089999994,
 124.40562920000002,
 74.28447259999984,
 169.0710224000004,
 94.02305089999936,
 82.20078309999917,
 74.78639629999998,
 78.22414130000016,
 61.79540250000082,
 78.87255780000032,
 73.19018479999977,
 84.52041720000034,
 91.35953510000036,
 68.08326529999977,
 63.8794269,
 90.41642629999933,
 165.02151390000017]
```

Python Query 6 Verification

Query 6

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
```

```
    where
        l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1994-01-01' + interval '1' year
        and l_discount between .06 - 0.01 and .06 + 0.01
        and l_quantity < 24;
```

In [37]:

```
q6_filter_date = datetime.datetime.strptime('1994-01-01', '%Y-%m-%d')
```

- They are same to the fifth decimal place.

In [38]:

```
# Python Result
q_06_python(lineitem_df, q6_filter_date)
```

Out[38]:

```
123141078.22829999
```

In [39]:

```
# TPC-H Result
q6_df = pd.read_csv(f"C:\ADM\Assignment1Output\q06.res.csv",header = None )
q6_df.values[0][0]
```

Out[39]:

```
123141078.2283
```

Python Query 6 Timings

SF1

In [41]:

```
func6 = %timeit -o -n 1 -r 20 q_06_python(lineitem_df, q6_filter_date)
func6.timings
```

```
845 ms ± 34.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

Out[41]:

```
[0.835034799999903,
 0.809838500000069,
 0.806849800000009,
 0.818629699999974,
 0.8260615000001508,
 0.8858207999999195,
 0.8162595999999667,
 0.8357582000001003,
 0.8263981999998578,
 0.8707277000000886,
 0.8562275000001591,
 0.858214500000031,
 0.8448459999999614,
 0.8281630999999834,
 0.8248092999999699,
 0.8470093999999335,
 0.8584554999999909,
 0.8372629999998935,
 0.8389979999999468,
 0.968523100000084]
```

SF3

In [71]:

```
func6_sf3 = %timeit -o -n 1 -r 20 q_06_python(lineitem_df3, q6_filter_date)
func6_sf3.timings
```

```
The slowest run took 8.92 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
15.1 s ± 8.13 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

Out[71]:

```
[41.36292830000002,
 22.83145129999957,
 19.130759800000305,
 10.755896499999835,
 20.469402900000205,
 20.78364689999944,
 13.561136100000112,
 12.981076899999607,
 10.69711820000066,
 16.451627099999314,
 9.5165793999995,
 6.20782539999982,
 4.636727699999938,
 5.108389800000623,
 11.146627699999954,
 11.375203700000384,
 13.610843599999498,
 10.053034700000353,
 17.306529800000135,
 23.214504400000806]
```

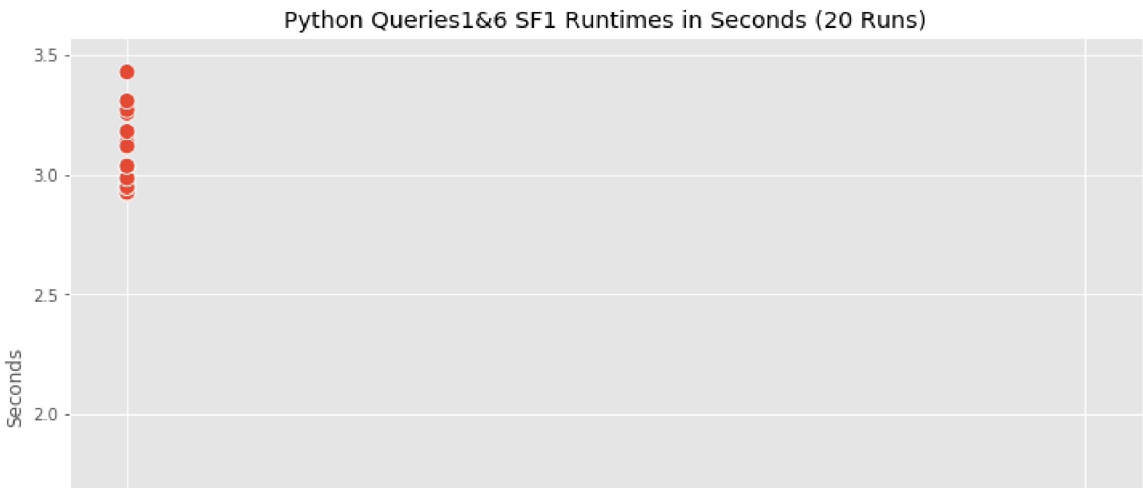
In [44]:

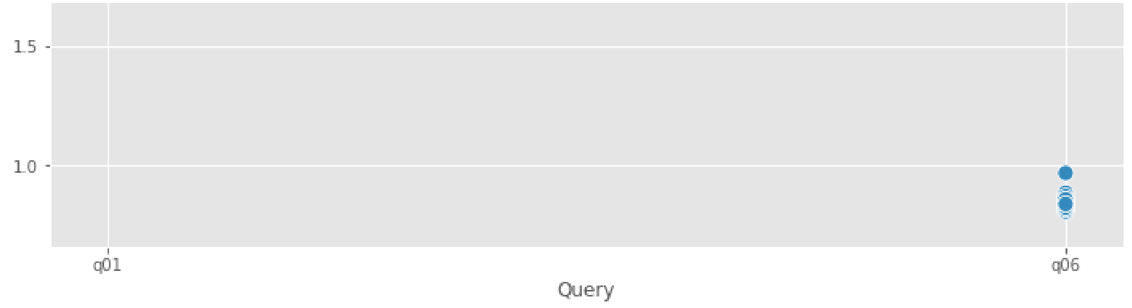
```
python_timing_dict = dict((q,0) for q in ["q01", "q06"])
python_timing_dict["q01"] = func1.timings
python_timing_dict["q06"] = func6.timings
python_timings_df = pd.DataFrame.from_dict(python_timing_dict)
pdfm = python_timings_df.melt(var_name='columns')
```

Python Queries Timings SF1 Visualised

In [46]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = pdfm, hue= 'columns', x = 'columns', legend = None, s =100 )
plt.title("Python Queries(1&6) SF1 Runtimes in Seconds (20 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```





In [49]:

```
python_timings_df
```

Out[49]:

	q01	q06
0	2.995881	0.835035
1	3.030532	0.809839
2	3.256670	0.806850
3	3.175232	0.818630
4	3.024105	0.826062
5	2.957718	0.885821
6	2.971614	0.816260
7	2.921598	0.835758
8	2.925873	0.826398
9	2.947681	0.870728
10	2.948778	0.856228
11	2.985393	0.858215
12	3.137876	0.844846
13	3.429341	0.828163
14	3.271930	0.824809
15	3.119845	0.847009
16	3.037133	0.858455
17	3.036601	0.837263
18	3.181289	0.838998
19	3.308941	0.968523

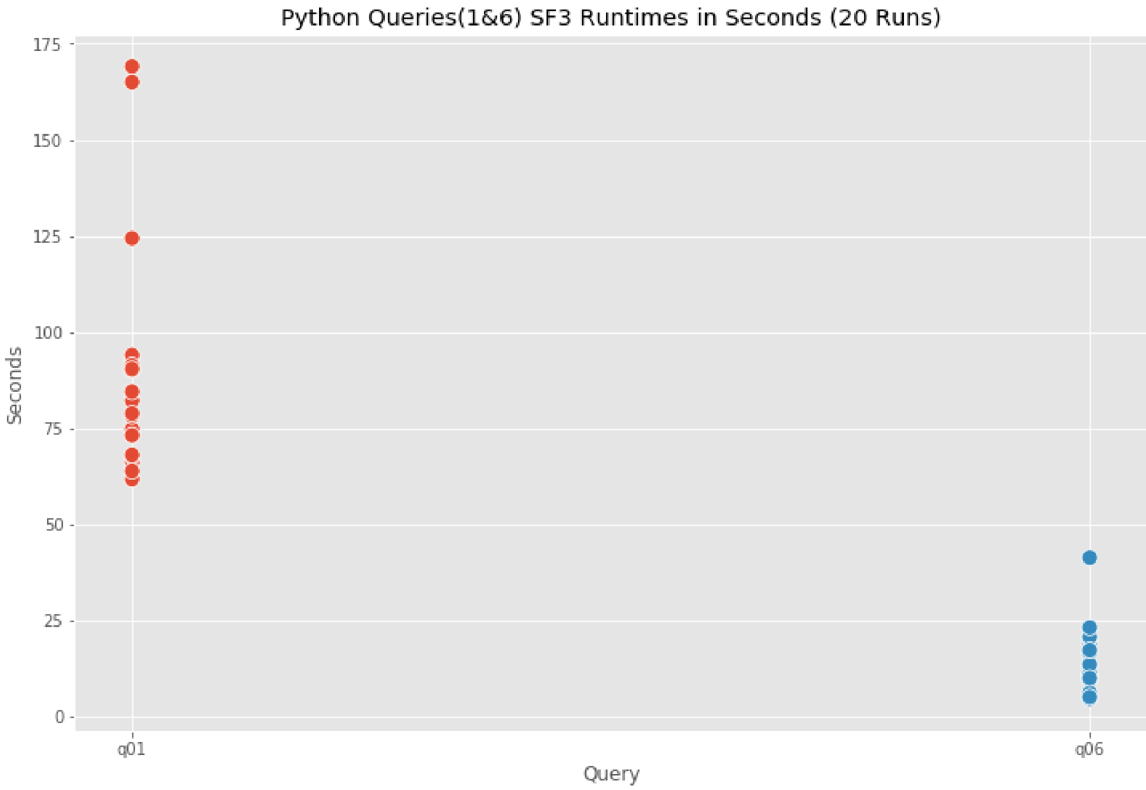
In [72]:

```
python_timing_dict_sf3 = dict((q,0) for q in ["q01", "q06"])
python_timing_dict_sf3["q01"] = func1_sf3.timings
python_timing_dict_sf3["q06"] = func6_sf3.timings
python_timings_df_sf3 = pd.DataFrame.from_dict(python_timing_dict_sf3)
pdfm_sf3 = python_timings_df_sf3.melt(var_name='columns')
```

Python Queries Timings SF3 Visualised

In [73]:

```
plt.rcParams["figure.figsize"]=12,8
sns.scatterplot(y = 'value', data = pdfm_sf3, hue= 'columns', x = 'columns', legend = None, s =100
)
plt.title("Python Queries(1&6) SF3 Runtimes in Seconds (20 Runs)")
plt.xlabel("Query")
plt.ylabel("Seconds")
plt.show()
```



Comparison & Conclusion

- **Fastest MonetDB, Slowest SQLite** - When looking across the distributions of run time across MonetDB, SQLite and Python, note that MonetDB is the fastest with the Python "queries" performing slighter faster (for Q1 & Q6) than with SQLite. More time would be needed to optimize the queries and the python code to compare fully.
- **Scaling to Larger Datasets** - As well as SQLite being generally slower than MonetDB, it does not scale as well as MonetDB does from SF1 to SF3 (which scales at a constant Big-O): The average query times run are > 3 times as high on the larger data set. Python "queries" did not scale well at all - e.g. each was 20-30 times slower on average on the SF3 data.
- **Choosing your Approach** - Looking at query (1&6) timings in more detail (below) it is clear than for these problems MonetDB performs best across the two SFs. Python performs significantly faster than SQLite with the smaller SF but it scales very poorly and with SF3 SQLite performs faster than Python.

Query One in Depth

```
MonetDB
SF1
    258 ms ± 79 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    786 ms ± 38.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)

Sqlite
SF1
    21.1 s ± 7.9 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    59.8 s ± 3.69 s per loop (mean ± std. dev. of 2 runs, 1 loop each)

Python
SF1
    3.08 s ± 142 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    1min 29s ± 29.3 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

Query Six in Depth

```
MonetDB
SF1
    19.3 ms ± 7.42 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    99.6 ms ± 7.22 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

```
    99.6 ms ± 7.22 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)

Sqlite
SF1
    3.01 s ± 49.9 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    9.09 s ± 79 ms per loop (mean ± std. dev. of 2 runs, 1 loop each)

Python
SF1
    845 ms ± 34.6 ms per loop (mean ± std. dev. of 20 runs, 1 loop each)
SF3
    15.1 s ± 8.13 s per loop (mean ± std. dev. of 20 runs, 1 loop each)
```

In [ ]:

```

```