Representative test query collection

WDQS is now in maintenance mode. We intend to check its health at regular intervals. To do this, we collect a small subset of sparql queries that represent all the sparql queries that come into WDQS. We run the small subset of queries at regular intervals of time and check how they perform, whether they are taking too long to response (even though they are typically fast queries), or are failing etc. This notebook tries to find what it means to *represent* all WDQS queries and how to collect a subset of them.

- What features of WDQS queries do we want represented? Response time, Query complexity etc.
- How many queries do we want containing each of the feature variations? Example: 10 queries with varying response times, 10 with varying query complexity etc. There may be overlapping features in queries as well.
- After we have chosen a set of queries, if we keep running them, the results may get cached and thus give us misguided information about the actual health of WDQS. To prevent caching we can perform the same queries with slight variation, like changing the items in the query.

In [ ]:

```
import wmfdata
import time
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.pyplot import text
import numpy as np
import pandas as pd
from statistics import mean
import seaborn as sns
from tqdm.notebook import tqdm
from IPython.display import clear_output
import plotly.express as px
import plotly
import matplotlib.pyplot as plt
from  matplotlib import animation
from matplotlib.animation import FuncAnimation
from IPython.display import HTML, Image, display, clear_output

spark = wmfdata.spark.get_session(type='yarn-large', app_name="wdqs-analysis",
                                  extra_settings={
                                            "spark.driver.memory": "32g",
                                            "spark.executor.memory": "32g",
                                            "spark.sql.shuffle.partitions": 1024
                                  }) #,ship_python_env=True

from pyspark.sql.functions import explode, col
import pyspark.sql.functions as F
```

Utils

In [2]:

```
## WIKIDATA
date = "20220801"
from_clause = f"from discovery.wikibase_rdf where date={date} and wiki='wikidata'"

wd = 'http://www.wikidata.org/entity/'
wdt = 'http://www.wikidata.org/prop/direct/'
p31 = wdt+'P31'

## SPARQL
date = [7,6,0] # month, day, hour

h_clause = f"year=2022 and month={date[0]} and day={date[1]} and hour={date[2]}"
d_clause = f"year=2022 and month={date[0]} and day={date[1]}"
m_clause = f"year=2022 and month={date[0]}"

from_sparql = f"from discovery.processed_external_sparql_query where {h_clause} and wiki='wikidata'"
from_sparql_d = f"from discovery.processed_external_sparql_query where {d_clause} and wiki='wikidata'"
from_sparql_m = f"from discovery.processed_external_sparql_query where {m_clause} and wiki='wikidata'"

event_wdqs = "event.wdqs_external_sparql_query"
```

- How many months/days/hours data to consider to gather representative queries from?
  - 1 month seems okay
- What features should we consider that should be represented in the small set of test queries?
  - response time
  - query structure (found from operator list)
  - query length
  - should we make sure various services are used even they aren't the *most* frequent queries?
  - Should we make sure to have some complex paths? (The larger the number of paths, possibly the more complex it is since a single complex path breaks down into all its individual path components)
  - Should we make sure the test queries have various kinds of expressions? (will be somewhat incorporated in opList)
- What happens when most queries are from a single UA or a bot? After query selection we need to make sure the UA/bots are varied for the queries.
- The idea is to select a few query *types* using the opList. For each query type we generate multiple queries by changing certain items. We need to make sure changing the items within the queries doesn't drastically change the time-distribution of the selected set of queries. Also distribution of the other features.

In [3]:

```
queries = spark.sql(f"select * {from_sparql_m}")
```

In [4]:

```
queries.printSchema()
```

```
root
 |-- id: string (nullable = true)
 |-- query: string (nullable = true)
 |-- query_time: long (nullable = true)
 |-- query_time_class: string (nullable = true)
 |-- ua: string (nullable = true)
 |-- q_info: struct (nullable = true)
 |    |-- queryReprinted: string (nullable = true)
 |    |-- opList: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |    |-- operators: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- prefixes: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- nodes: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- services: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- wikidataNames: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- expressions: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- paths: map (nullable = true)
 |    |    |-- key: string
 |    |    |-- value: long (valueContainsNull = true)
 |    |-- triples: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- subjectNode: struct (nullable = true)
 |    |    |    |    |-- nodeType: string (nullable = true)
 |    |    |    |    |-- nodeValue: string (nullable = true)
 |    |    |    |-- predicateNode: struct (nullable = true)
 |    |    |    |    |-- nodeType: string (nullable = true)
 |    |    |    |    |-- nodeValue: string (nullable = true)
 |    |    |    |-- objectNode: struct (nullable = true)
 |    |    |    |    |-- nodeType: string (nullable = true)
 |    |    |    |    |-- nodeValue: string (nullable = true)
 |-- year: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- day: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- wiki: string (nullable = true)
```

In [5]:

```
oplist_grp = queries.groupby("q_info.opList").count().sort("count", ascending=False)
```

In [17]:

```
## how many different opList (operator list) are there, this represents the query structure
oplist_grp.count()
```

Out[17]:

```
20751
```

In [6]:

```
## See the top query structures, or opList
oplist_grp.show(10, truncate=False)
```

```
[Stage 2:====================================================>(1016 + 8) / 1024]
+-------------------------------------------------------------------------------------+--------+
|opList                                                                               |count   |
+-------------------------------------------------------------------------------------+--------+
|[bgp, bgp, service, join, project, slice]                                            |83872722|
|[bgp, bgp, service, join, project]                                                   |51581658|
|[path, table, bgp, join, bgp, union, join, project]                                  |13562846|
|[table, bgp, join, bgp, leftjoin, bgp, service, join, extend, extend, order, project]|10911230|
|[table, bgp, join, bgp, service, join, project, distinct]                            |10588895|
|[path, table, bgp, join, bgp, union, bgp, union, join, project]                      |10541148|
|[bgp]                                                                                |9931820 |
|[bgp, project, distinct]                                                             |8371104 |
|[bgp, project, distinct, slice]                                                      |7765898 |
|[table, bgp, join, filter, project, distinct]                                        |7281862 |
+-------------------------------------------------------------------------------------+--------+
only showing top 10 rows
```

In [19]:

```
time_grp = queries.groupby("query_time_class").count().sort("count", ascending=False)
```

In [20]:

```
## showing the various time_groups we have for all queries
time_grp.show(100)
```

```
[Stage 14:===================================================>(1021 + 3) / 1024]
+----------------+---------+
|query_time_class|    count|
+----------------+---------+
| 2_10ms_to_100ms|283557726|
|   3_100ms_to_1s| 59934451|
|     1_less_10ms|  7904073|
|     4_1s_to_10s|  3730660|
|      5_more_10s|  1776827|
+----------------+---------+
```

In [26]:

```
## Number of disinct opList per time group
queries.groupby("query_time_class").agg(F.countDistinct("q_info.opList").alias("distinct_opList")).show(100)
```

```
+----------------+---------------+
|query_time_class|distinct_opList|
+----------------+---------------+
|     4_1s_to_10s|          10329|
|   3_100ms_to_1s|          11347|
| 2_10ms_to_100ms|           5718|
|      5_more_10s|           7900|
|     1_less_10ms|            421|
+----------------+---------------+
```

In [7]:

```
pd.set_option('display.max_colwidth', None)
```

Automate query selection

Manually selecting query types (not feasible)

- Queries are grouped by query_time_group and the opList (operator list), the number of distinct queries per group were counted.
- For each group, we look at simple yet most occuring queries. Simple queries are those that don't have too many operators in the opList. But we look for diverse types of queries, such as, some have project, filters, table, order by, paths etc.
- We select time groups upto that take upto 1 second. The next group takes 1s to 10s, and those queries may be too long to be test queries. So we choose the first three time groups.
- Next we look at and select individual queries from each group of query types

Automatic process to select queries:

- The demerit of picking up query types individually is that
  - it is time consuming
  - selecting query types is not trivial and doesn't guarantee that we chose a good set of query types
- To do this relatively automatically
  - We can select the first 10-15 query types for each query_time_class.
  - We can then remove any query type that seems too long (e.g those with too many joins). We may even decide to keep those queries, but select less of them.
  - For each query type group, we can save a set of 50 queries. To run test queries we can select 1 or 2 from the pool of 50 queries from each query type group. This will also alleviate the need to 'generate' queries by changing items in the queries, because we already have a large of queries to choose from in each query type group.
- Something to keep in mind: we are running the automation for a months data. We could run it for longer data to get better representation. We can also run the script at some intervals of time, such as every 6 months and save a new set of queries that we can test with.

In [4]:

```
from pyspark.sql.window import Window

## select 15 query type group per query time class
w = Window.partitionBy("query_time_class").orderBy(F.col("count").desc())
time_oplist_grp = queries.groupby(["query_time_class", "q_info.opList"]).agg(F.countDistinct("q_info.queryReprinted").alias("count"))
top_time_oplist_grp = time_oplist_grp.withColumn("row",F.row_number().over(w)).filter(F.col("row") <= 15).drop("row")\
                                     .sort(F.col("query_time_class").asc(), F.col("count").desc())

## Getting the top queries for all time classes at once ran out of spark time and memory. So each of the
## 3 time classes were listed separately and then the final results were merged.
w = Window.partitionBy(["query_time_class", "q_info.opList"]).orderBy(F.lit("A"))

top_time_oplist_grp_1 = top_time_oplist_grp.filter("query_time_class = '1_less_10ms'").drop("query_time_class")
top_time_oplist_grp_2 = top_time_oplist_grp.filter("query_time_class = '2_10ms_to_100ms'").drop("query_time_class")
top_time_oplist_grp_3 = top_time_oplist_grp.filter("query_time_class = '3_100ms_to_1s'").drop("query_time_class")

queries_1 = queries.filter("query_time_class = '1_less_10ms'")
queries_2 = queries.filter("query_time_class = '2_10ms_to_100ms'")
queries_3 = queries.filter("query_time_class = '3_100ms_to_1s'")

## select 50 random queries per group
test_queries_1 = queries_1.join(top_time_oplist_grp_1, queries_1.q_info.opList == top_time_oplist_grp_1.opList, "inner")\
         .withColumn("row",F.row_number().over(w)).filter(F.col("row") <= 50).drop("row")

test_queries_2 = queries_2.join(top_time_oplist_grp_2, queries_2.q_info.opList == top_time_oplist_grp_2.opList, "inner")\
         .withColumn("row",F.row_number().over(w)).filter(F.col("row") <= 50).drop("row")

test_queries_3 = queries_3.join(top_time_oplist_grp_3, queries_3.q_info.opList == top_time_oplist_grp_3.opList, "inner")\
         .withColumn("row",F.row_number().over(w)).filter(F.col("row") <= 50).drop("row")
```

In [12]:

```
test_queries_1.count()
```

```
[Stage 12:===============>(47 + 1) / 50][Stage 14:===========>(1615 + 5) / 1620]22/08/15 13:41:24 WARN TaskSetManager: Lost task 4.0 in stage 12.2 (TID 10287, an-worker1087.eqiad.wmnet, executor 260): TaskKilled (Stage cancelled)
```

Out[12]:

```
750
```

In [13]:

```
test_queries_2.count()
```

Out[13]:

```
750
```

In [16]:

```
test_queries_3.count()
```

Out[16]:

```
750
```

In [5]:

```
## merge queries from all time classes
test_queries = test_queries_1.union(test_queries_2).union(test_queries_3)
test_queries.cache()
```

Out[5]:

```
DataFrame[id: string, query: string, query_time: bigint, query_time_class: string, ua: string, q_info: struct<queryReprinted:string,opList:array<string>,operators:map<string,bigint>,prefixes:map<string,bigint>,nodes:map<string,bigint>,services:map<string,bigint>,wikidataNames:map<string,bigint>,expressions:map<string,bigint>,paths:map<string,bigint>,triples:array<struct<subjectNode:struct<nodeType:string,nodeValue:string>,predicateNode:struct<nodeType:string,nodeValue:string>,objectNode:struct<nodeType:string,nodeValue:string>>>>, year: int, month: int, day: int, hour: int, wiki: string, opList: array<string>, count: bigint]
```

In [ ]:

```
test_queries.select("id", "query", "ua", "query_time_class", "opList").toPandas().to_csv("test_queries.csv", index=False)
```

Analysis on Listed Queries

Check UA

In [20]:

```
ua = test_queries.groupby("ua").count()
ua.count()
```

Out[20]:

```
366
```

Check Services used

In [30]:

```
services = test_queries.select("*", F.explode("q_info.services").alias("service", "_"))\
                .groupby("service").agg(F.countDistinct("id").alias("count"))
```

In [31]:

```
## List of services used in the selected queries and number of queries that uses each service
services.sort("count", ascending=False).show(100, truncate=False)
```

```
[Stage 300:==================================================>(1022 + 2) / 1024]
+-------------------------+-----+
|service                  |count|
+-------------------------+-----+
|NODE_URI[wikibase:label] |800  |
|NODE_URI[wikibase:mwapi] |250  |
|NODE_URI[wikibase:around]|50   |
+-------------------------+-----+
```

In [4]:

```
## List and count of all service usage in all queries (for a month).
## This is listed to compare with the previous list which is only for the selected queries.
queries.select("*", F.explode("q_info.services").alias("service", "_"))\
        .groupby("service").agg(F.countDistinct("id").alias("count"))\
        .sort("count", ascending=False).show(20, truncate=False)
```

```
[Stage 2:====================================================>(1020 + 4) / 1024]
+--------------------------------------------------------------------------+---------+
|service                                                                   |count    |
+--------------------------------------------------------------------------+---------+
|NODE_URI[wikibase:label]                                                  |204150704|
|NODE_URI[wikibase:mwapi]                                                  |18326814 |
|NODE_URI[wikibase:around]                                                 |3133220  |
|NODE_URI[gas:service]                                                     |209038   |
|NODE_URI[wikibase:box]                                                    |189102   |
|NODE_URI[bd:slice]                                                        |108905   |
|NODE_URI[http://dbpedia.org/sparql]                                       |17240    |
|NODE_URI[https://query.wikidata.org/sparql]                               |17184    |
|NODE_URI[bd:sample]                                                       |1262     |
|NODE_URI[http://data.cervantesvirtual.com/openrdf-sesame/repositories/data]|81       |
|NODE_URI[mediawiki:categoryTree]                                          |56       |
|NODE_URI[http://sparql.europeana.eu/]                                     |51       |
|NODE_URI[https://query.wikidata.org/bigdata/namespace/categories/sparql]  |44       |
|NODE_URI[http://data.bnf.fr/sparql]                                       |42       |
|NODE_URI[http://zbw.eu/beta/sparql/pm20/query]                            |36       |
|NODE_URI[http://sparql.wikipathways.org/sparql]                           |30       |
|NODE_URI[http://data.nobelprize.org/sparql]                               |21       |
|NODE_URI[https://dbpedia.org/sparql]                                      |18       |
|NODE_URI[https://data.pdok.nl/sparql]                                     |18       |
|NODE_URI[http://datos.bne.es/sparql]                                      |18       |
+--------------------------------------------------------------------------+---------+
only showing top 20 rows
```

In [ ]:

In [ ]: