# Combining SQL and Python to Create Runtime Performance Optimizations on TPC-H OLAP Queries in SQLite3

Kevin Fortier, Jonathan Duran and William Campbell

## Abstract online

The TPC-H benchmark was created as a standard to measure the performance of database management systems. The TPC-H benchmark consists of 22 queries that typify Online Analytical Processing queries found in various industries. In this work here, we set out to measure the performance of each of the 22 queries on a SQLlite3 database, and to classify the queries based on their runtime compared to one another. We then used a general purpose programming language in addition to the SQLite database to optimize one of the slow running query. This hybrid approach to query optimization is a reproducible pattern that can be generalized to optimize many types of queries where the underlying database system is unable to easily optimize a given query. The code for this project can be found here: https://github.com/kforti/TPCH-Benchmark-sqlite3-python [9].

## Introduction

Database benchmarks play an important role in the measurement and improvement of database performance. They are standards formed and created by organizations whose goal it is to create a way to measure a database management systems performance in a standardized way. These benchmarks are meant to be capable of characterizing and comparing the performance (in terms of time, memory, or quality) of database systems and/or algorithms executed on those systems in a reproducible way [6]. One of the most widely used benchmarks is TPC-H, created by the Transaction Processing Performance Council (TPC). The TPC is a non-profit organization founded in 1988 with the goal of defining transaction processing and database benchmarks and to disseminate these benchmarks to the industry [5]. The TPC-H benchmark is a "decision support benchmark" containing a set of ad hoc, business oriented queries and concurrent data modifications [1]. According to the TPC-H homepage, "the queries and the data populating the database have been chosen to have broad industry-wide relevance." [1]

TPC-H is considered an OLAP benchmark, which stands for Online Analytical Processing [3]. Contained within the TPC-H benchmark are 22 queries that are meant as a representative set of queries that typify the types of workloads performed in database systems within various industries [10]. Along with the definition of these 22 queries are validation parameters and results to be used for ensuring proper functioning of these queries on the database system in question [10]. Some of these queries contain intentionally inefficient operations, often referred to as “choke points”. The TPC-H benchmark rewards those database systems with the ability to optimize these inefficient queries, as this illustrates the system’s overall ability to handle the underlying technical challenge that the inefficient query is meant to represent.

While some database systems may not have built-in solutions for the choke points introduced in the TPC-H benchmark, or any database for that matter, modern OLAP query and data analysis environments often have a wide range of additional tooling that work alongside the main query language. For example, data analysts may execute certain queries in the standard SQL query language but may perform downstream analysis in a general programming language like python. Our observation is that a general programming language can be used to offer certain optimizations that are not supported by the underlying database system.

The python programming language is a popular general purpose programming language because of its simple syntax and large community of contributors as well as software packages.

Python is an ideal language to use for the purpose of combining with SQL for optimizations because it is widely used for these types of data processing jobs both within and outside the context of database queries.

Our goal in this work is to execute the 22 queries specified in the TPC-H benchmark in a programmatic way using the python programming language and SQLite3, and measure the runtime performance of each query and use this performance data to classify the queries in terms of their speed performance. Finally, we use our performance metrics to better understand specific choke points and create an optimization solution utilizing both the SQLite3 and python.

## Methods/Approach

### Measuring Performance

To measure the performance of each of the 22 queries mentioned in the TPC-H benchmark we executed each of the queries and measured the runtime with python. The queries were stored as string variables in python and executed with the python SQLite3 library. For each executed query we saved the results returned and the runtime to files on disk. We used the returned records from the query execution to compare to the validation output in the TPC-H benchmark to ensure the correctness of our queries, and we used the runtime date we gathered to compare the query performance across each of the 22 queries.

### Code Architecture

As was previously mentioned, our queries were executed with python. Python has a built-in module for interacting with SQLite3. We created a pattern for defining queries, rendering the query as a string and executing the query all within a python program. The pattern we developed was reusable across all 22 queries and proved to be extendable as well because we ultimately implemented an optimized query that leveraged additional python packages using the same design pattern.

Briefly, our code's design pattern used a base query class with 2 methods: get_query and run. As the name implies, get_query is used to get the query- in the base class we are just returning a predefined query as a string, which must be defined when the query class is defined, but the query could include more logic to be dynamically built using variables passed into the query class during its construction. The run method contains code for executing a query retrieved from the get_query method on a database while also measuring the time of executing the query. The run method returns the result of the query and the runtime metric. All of the queries we defined inherited from this base class and when necessary, these main methods were overwritten to meet the requirements of the specific query.

Our system also involves another function to execute a collection of queries. This function takes a collection of queries, executes each one and saves the rows returned to a file and stores the name of the query along with the runtime for that query. The final output is a directory with a file of results for each executed query and a csv file with the query name and runtime as the two columns.

Architecting our system in this way makes it simple to selectively run and rerun queries. The definition and execution of each query is separate from the higher-level logic for executing all queries and analyzing the performance metrics.

Finally, our code structure also consists of several short scripts for analyzing the performance metrics of the queries, as well as the queries themselves. In other words, we attempted to use python to parse the SQL queries to identify keywords, entities and attributes used

in each query as well as the frequency with which they were used- more on this the discussion section.

### Query Optimizations

We used the pandas and numpy python libraries in order to perform query optimizations in cases where SQLite3 could not optimize the original query described in the TPC-H benchmark [11 & 12]. Performance of the optimized query was done in the same way as described above, except now there is additional python code where further optimizations are done. The execution of the python code is considered when measuring the runtime of the query.

## Results

Each of the 22 queries were measured for their overall time to completion, as described above. Table 1 shows the raw results of the time to completion for each query. In order to better visualize the performance of the queries despite the large discrepancies seen in the raw data, we plotted the log base 2 of the runtime for each query, as seen in Figure 1.

After visualizing the performance of each query as well as performing a basic statistical analysis (seen in Table 2), we then chose to create an optimized version of query 17 using a hybrid of SQL and python. The final result returned from the execution of the optimized version of query 17 matched the original query 17 as well as the validation result specified in the TPC-H benchmark. The optimized query 17 executed in less than 15 seconds compared to the 4222 seconds it took for the original query 17.

## Discussion

Based on the raw performance results shown in Figure 1, it can clearly be seen that query 17, 20 and 22 all ran significantly slower than the other 19 queries. Our initial goal was to create bins or classifications to put the queries in and to try to identify some unique characteristics of the queries that placed them into their respective bin. For example, we tried parsing out the entities, attributes and SQL keywords from the queries as well as their frequency. However, no immediate patterns were evident from this type of analysis. Instead, since some queries had much longer runtimes compared to others, we decided to classify the queries as either simply either slow or fast and we then worked to apply our hybrid optimization strategy to query 17 by addressing some of the choke points identified in this query by [8].

To address the slow runtime of query 17, we tried to remove the correlated subquery while also pushing the predicate logic down from the outer join- these were two of the optimizations suggested by [8]. In query 17, both the outer join and the subquery join involve joining the ‘lineitem’ and ‘part’ tables, and the predicate logic in the outer join can occur in the subquery. According to [8], the TPC-H benchmark rewards systems that can buffer the intermediate result of the join and not have to perform this join on each row return from the outer query. We accomplished this optimization in our hybrid query system by performing a single join of the ‘lineitem’ and ‘part’ tables with all the predicate logic found in the original outer join. We then perform the logic involving the subquery and the original query project in python.

More specifically, our python code for query 17 involved transforming the records outputted by SQLite3 into a pandas dataframe, calculating the average price per part and then selecting the records whose quantity was less than 20% of the average for that part. Finally, we summed the ‘extendedprice’ attribute and divided it by 7 to get the per year average over the 7 year history within the database.

The pattern that we used for defining queries in both SQL and python can be used to define many additional types of queries including some of the others identified in the TPC-H benchmark, we discuss these other queries later in this section. Another possibility with this hybrid approach is to apply our earlier analysis of parsing the SQL query to identify the underlying structure of the queries. Prebuilt python functions could be built and applied automatically to queries based on a static analysis of the query structure within the python programming language. This would allow for automatic optimizations of certain types of queries. Our suggested approach to building such automatic optimizations is to start with simple heuristics in the static analysis, e.g. ‘is there a subquery?’ ‘is there predicate logic in the outer query?’ ’Does the outer query and subquery involve the same entities and operations?’. From these simple static analysis heuristics one could develop simple query optimizations to be applied in conjunction with the original query or the original query could be programmatically altered in some way.

In addition to optimizing query 17, the next steps for this project could be to also come up with python-based optimizations for queries 20 and 22. Next, we will briefly describe these queries and how we would go about optimizing them.

Query 20 involves a correlated subquery that contains a join. This query could be modified in a similar fashion to our optimized query 17. Query 22 also contains correlated subqueries, in addition to complex expression logic. This query could possibly be broken down into multiple queries executed in SQL in order to reduce the initial result set that some of the complex expressions are operating on. In this way, the database system will not be required to compute complex expression logic across each row. For example, there is a big IN predicate against a set of multiple values. By applying the solution of an “invisible join”, where a hash-table is used for converting the predicate into a semi-join, we could reduce the runtime [8] .

During our experimentation and running of the 22 TPC-H queries we encountered results that did not match the validation for 2 of the 22 queries. For example, query 2 returned a list of records in ascending order starting with records that had a negative account balance value. However, the validation result in the TPC-H documentation shows a sample output record with a large positive value as the account balance. It could be the case that the results attained by the TPC-H documentation returned records in descending order. We could not confirm this hypothesis with the original query because there is a limit of 100 set on the number of records returned. Another query we encountered problems with is query 6. Our query returned a different result compared to the benchmark documentation. We could not determine why this query was wrong.

## Conclusion

We were able to measure the runtimes for each query, and group them based on their performance relative to one another. We were able to optimize one long-running query in depth and develop a reusable pattern for applying query optimizations when the underlying database system is unable to optimize the query. A major takeaway from our investigation into the TPC-H benchmark and its 22 queries is the role a general purpose language like python can play in the optimization process. If the execution of the 22 queries is limited strictly to the capabilities of the underlying database, they are forced to run under its capabilities and in accordance with how the query was input. With access to additional tooling, as is often the case in modern OLAP query environments, we're able to restructure how the queries get processed utilizing more than just the query language and capabilities of the database system.

## Figures and Tables

**Table 1.**

| Name | Time to Completion |
|---|---|
| query_1 | 4.287 |
| query_2 | 0.318 |
| query_3 | 1.916 |
| query_4 | 2.528 |
| query_5 | 2.829 |
| query_6 | 1.308 |
| query_7 | 4.229 |
| query_8 | 6.288 |
| query_9 | 16.488 |
| query_10 | 1.539 |
| query_11 | 0.781 |
| query_12 | 1.321 |
| query_13 | 11.496 |
| query_14 | 1.488 |
| query_15 | 2.924 |
| query_16 | 0.459 |
| query_17 | 4222.523 |
| query_18 | 1.808 |
| query_19 | 1.957 |
| query_20 | 5664.344 |
| query_21 | 9.129 |
| query_22 | 1086.091 |

**Table 1,** shows the raw results of the completion time for each of the 22 queries defined in the TPC-H benchmark with the validation parameters defined in the query.

**Table 2.**

| | |
|---|---|
| mean | 502.093 |
| std | 1472.745 |
| min | 0.318 |
| 25% | 1.501 |
| 50% | 2.679 |
| 75% | 8.419 |
| max | 5664.344 |

**Table 2,** shows the mean, standard deviation, minimum value, 25th percentile, 50th percentile, 75th percentile and the maximum value for the runtime of the 22 queries defined in the TPC-H benchmark.

**Figure 1.**

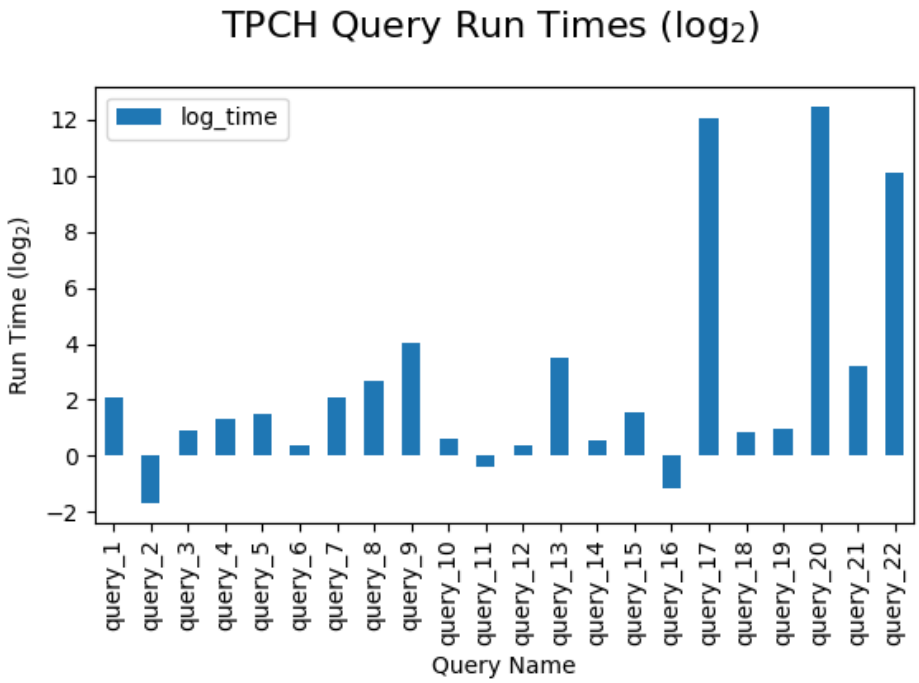


**Figure 1,** shows the $\log_2$ of the time to completion for each of the 22 queries defined in the TPC-H benchmark.

**Figure 2.**

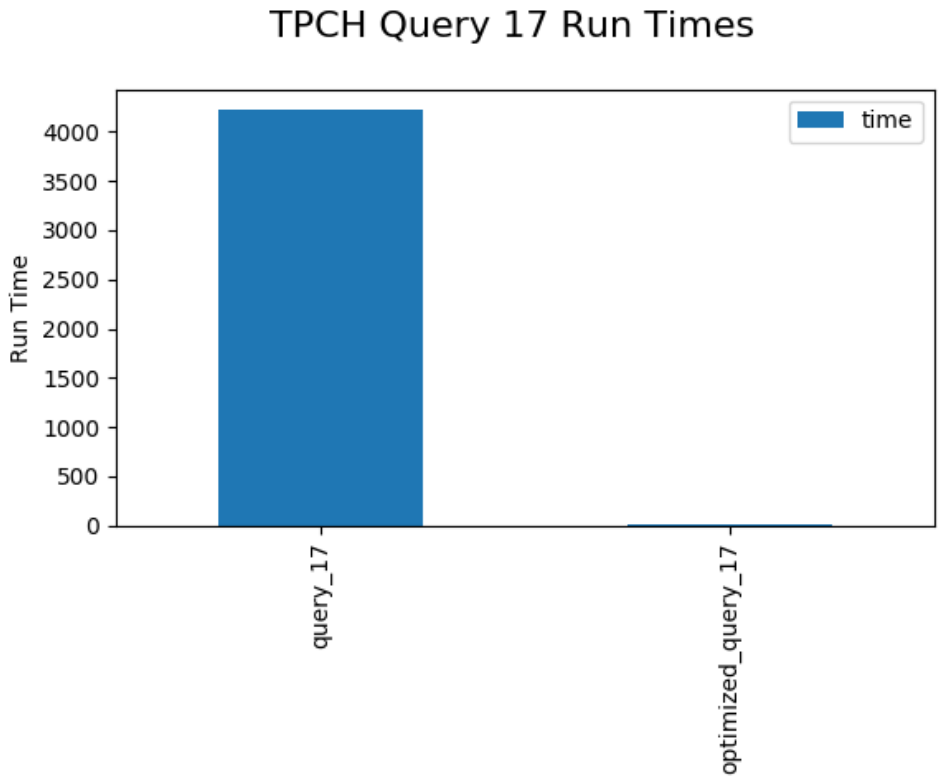


**Figure 2,** shows the time to completion for original query 17 from the TPC-H defined in the SQL and the optimized version of query 17 presented in this paper.

## References


1. TPC-H Homepage. http://www.tpc.org/tpch/
2. TPC-H Query Plans. http://www.qdpma.com/tpch/TPCH100_Query_plans.html
3. Meyen, Nuzhi. “Understanding TPC-H and TPC-E Performance Benchmarks.” Medium, 9 July 2020, https://nmeyen.medium.com/understanding-tpc-h-and-tpc-e-performance-benchmarks-d6da686dc769
4. “What Is A TPC-H Benchmark? Data Defined.” Indicative, 20 Feb. 2020, https://www.indicative.com/data-defined/tcp-h-benchmark/
5. “10 Questions: The TPC-H Benchmark.” Exasol, https://www.exasol.com/resource/10-questions-about-the-tpc-h-benchmark/
6. Bonnet, Philippe, and Dennis Shasha. “Database Benchmarks.” Encyclopedia of Database Systems, edited by Ling Liu and M. Tamer Özsu, Springer, 2018, pp. 936–38. Springer Link, doi:10.1007/978-1-4614-8265-9_80797
7. TPC Who We Are. http://tpc.org/information/who/whoweare5.asp
8. Peter Boncz, Thomas Neumann, and Orri Erling, "TPC-H Analyzed: Hidden Messages and LessonsLearned from an Influential Benchmark"
9. Github Repository for this Project, https://github.com/kforti/TPCH-Benchmark-sqlite3-python
10. TPC Benchmark H Standard Specification, Revision 3.0.0. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf
11. Pandas - Python Data Analysis Library. https://pandas.pydata.org/
12. NumPy. https://numpy.org/