Investigation of TPC-H Benchmark

Romaji Amulo, Kimberly Brady, Vinit Kothari, Joshua Malcarne

Database Management Systems, Worcester Polytechnic Institute

Final Project
27 April 2022

Table of Contents

List of Figures

List of Tables

1. Introduction

The collection, management, and processing of data has become an essential part of the United States' economy. National daily data usage rose from approximately 12 GB in 2019 to 16.6 GB in 2020, an increase of 38% in just one year (Blue Bird Network, 2021). This increase was spread across all industries, and the success of most businesses now relies on access to relevant data. To manage large quantities of data, which is often unstructured and unusable in its raw form, businesses need methods to store and manage their data. Database management systems (DBMS) are software platforms used to manage such data. DBMS are used globally, and include many widely used applications, such as Amazon purchasing systems, bank transactions, and Google Maps. Multiple users can concurrently and securely interact with data through these systems and can perform actions, such as storing, querying, indexing, and structuring data, while maintaining data integrity (Raza & Wickramasinghe, 2021). Due to the importance and complexity of DBMS, their performance and scalability must be understood.

Query execution is a vital part of DBMS. Queries go through many steps to produce the intended output; these steps are referred to as the query lifecycle. First, the written query is converted to a parse tree, which is then converted to a logical query plan. The logical query plan can be re-written (once or many times) to maximize efficiency, which is estimated by applying certain statistics (e.g., number of tuples in a relation). For each logical plan, multiple physical plans (e.g., one-pass, two-pass, or multi-pass algorithms) can be developed, each which have an associated cost. Finally, after calculating the costs of each physical plan, the best plan is selected and the query is executed. Database optimizers work to optimize this query lifecycle to produce the desired results in the shortest amount of time with the lowest cost. DBMS with an efficient query optimizer and query execution engine can help save companies millions of dollars (Chaudhuri, 1998).

Benchmarking is a proven and clearly defined method for comparing and characterizing the performance of DBMS using a set of pre-defined queries. Benchmarking can help companies make informed decisions about which DBMS to purchase for their needs. The Transaction Processing Performance Council (TPC) was formed to define benchmarks (e.g., the TPC-H decision-support benchmark) for transaction processing and database domains (Benchant, 2022) and is widely used. According to TPC.org, the TPC-H benchmark examines large volumes of data, executes complex queries (written in the Structured Query Language (SQL)), and answers critical business questions. Each query is defined by the business question, functional query definition, substitution parameters, and query validation, and uses the Composite Query-per-Hour Performance Metric (QphH@Size) (TPC-H.org, 2022).

2. Methods, Data, and Results

In the following sections, we describe details about the TPC-H system and analyze the pre-written queries in different ways. In Section 2.1, we examine the data types and tables that are part of each individual query. In Section 2.2, we compare the runtimes for each query using three different machines, and in Section 2.3, we take a closer look at ways to classify the queries based on certain characteristics, such as the runtimes. In Section 2.4, we discuss the potential for optimization for select queries, and in Section 2.5 we briefly describe correlations between columns.

2.1 Overview of Data Types and Tables for Each Query

The TPC-H benchmark has 8 tables, 8661245 rows, 61 columns, and real-valued, discrete, and categorical data types. Table 1 lists the different data types included in each query.

Table 1: Data types for each query (source: TPC, 2021)

Column Name	Datatype	Requirements
PART Table (Primary Key: P_PARTKEY) Q2,8,9,14,16,17,19,20		
P_PARTKEY	identifier	SF*200,000 are populated
P_NAME	variable text, size 55	

Column Name	Datatype	Requirements
P_MFGR	fixed text, size 25	
P_BRAND	fixed text, size 10	
P_TYPE	variable text, size 25	
P_SIZE	integer	
P_CONTAINER	fixed text	
P_RETAILPRICE	decimal	
P_COMMENT	variable text	
SUPPLIER Table (Primary Key: S_SUPPKEY) Q2,5,7,8,9,11,15,20,21		
S_SUPPKEY	identifier	SF*10,000 are populated
S_NAME	fixed text, size 25	
S_ADDRESS	variable text	
S_NATIONKEY	Identifier, size 40	
S_PHONE	fixed text, size 15	
S_ACCTBAL	decimal	
S_COMMENT	variable text, size 101	
PARTSUPP Table (Primary Key: PS_PARTKEY, PS_SUPPKEY) Q2,9,11,16,20		
PS_PARTKEY	Identifier	Foreign Key to P_PARTKEY
PS_SUPPKEY	Identifier	Foreign Key to S_SUPPKEY
PS_AVAILQTY	integer	
PS_SUPPLYCOST	Decimal	
PS_COMMENT	variable text, size 199	
CUSTOMER Table (Primary Key: C_CUSTKEY) Q3,5,7.8,10,13,18,22		
C_CUSTKEY	Identifier	SF*150,000 are populated
C_NAME	variable text, size 25	
C_NATIONKEY	Identifier	Foreign Key to N_NATIONKEY
C_PHONE	fixed text, size 15	
C_ACCTBAL	Decimal	
C_MKTSEGMENT	fixed text, size 10	
C_COMMENT	variable text, size 117	
ORDERS Table* (Primary Key: O_ORDERKEY) Q3,4,5,7,8,9,10,12,13,18,21,22		
O_ORDERKEY	Identifier	SF*1,500,000 are sparsely populated
O_CUSTKEY	Identifier	Foreign Key to C_CUSTKEY
O_ORDERSTATUS	fixed text, size 1	
O_TOTALPRICE	Decimal	
O_ORDERDATE	Date	
O_ORDERPRIORITY	fixed text, size 15	
O_CLERK	fixed text, size 15	
O_SHIPPRIORITY	Integer	
O_COMMENT	variable text, size 79	
LINEITEM Table (Primary Key: L_ORDERKEY, L_LINENUMBER) Q1,3,4,5,6,7,8,9,10,12,14,15,17,18,19,20,21		
L_ORDERKEY	identifier	Foreign Key to O_ORDERKEY
L_PARTKEY	identifier	Foreign key to P_PARTKEY, first part of the compound Foreign Key to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY	Identifier	Foreign key to S_SUPPKEY, second part of the compound Foreign Key to (PS_PARTKEY, PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER	integer	
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_TAX	decimal	
L_RETURNFLAG	fixed text, size 1	
L_LINESTATUS	fixed text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	fixed text, size 25	
L_SHIPMODE	fixed text, size 10	

Column Name	Datatype	Requirements
L_COMMENT	variable text size 44	
NATION Table (Primary Key: N_NATIONKEY) Q2,5,7 (n1 and n2),8 (n1 and n2),9,10,11,20,21		
N_NATIONKEY	identifier	25 nations are populated
N_NAME	fixed text, size 25	
N_REGIONKEY	identifier	Foreign Key to R_REGIONKEY
N_COMMENT	variable text, size 152	
REGION Table (Primary Key: R_REGIONKEY) Q2,5,8		
R_REGIONKEY	identifier	5 regions are populated
R_NAME	fixed text, size 25	
R_COMMENT	variable text, size 152	

Data type descriptions (TPC-H.org, 2022):

- Identifier: holds any key value generated for that column and supports at least 2,147,483,647 unique values
- Integer: exactly represents integer values (i.e., values in increments of 1) in the range of at least -2,147,483,646 to 2,147,483,647
- Decimal: represents values in the range -9,999,999,999.99 to +9,999,999,999.99 in increments of 0.01; the values can be either represented exactly or interpreted to be in this range
- Fixed text, size N: holds any string of characters of a fixed length of N
- Variable text, size N: holds any string of characters of a variable length with a maximum length of N
- Date: a value expressed as YYYY-MM-DD with all numeric characters

Table 2 shows the tables included in each query and the different data types used in the Select Clause in the query (note, FT=fixed text; Dec=decimal; Ident=identifier; VT=variable text).

Table 2: Tables included in each query

Query	Part	Supplier	Partsupp	Customer	Orders	Line Item	Nation	Region	Data Types in Select Clause
1						x			FT, Dec
2	x	x	x				x	x	FT, Dec, Ident, VT
3				x	x	x			Ident, Dec
4					x	x			FT
5		x		x	x	x	x	x	FT, Dec
6						x			Dec
7		x		x	x	x	x		FT, Date, Dec
8	x	x		x	x	x	x	x	Date, FT, Dec
9	x	x	x		x	x	x		FT, Date, Dec
10				x	x		x		Ident, FT, Dec, VT
11		x	x				x		Ident, D, Integer
12					x	x			FT
13					x				Ident
14	x					x			VT, Dec
15		x				x			Ident, FT, VT
16	x		x						FT, VT, Integer, Ident
17	x					x			Dec
18				x	x	x			VT, Ident, Date, Dec
19	x					x			Dec
20	x	x	x			x	x		FT, VT, Ident, Dec
21		x			x	x	x		FT, Ident, Integer, Dec, Date, VT
22				x	x				FT, Dec, Ident, Date, Integer, VT

The table accessed by the most queries is the LineItem table, followed by Orders (both are highlighted in green). The least accessed table is Region, followed by PartSupp (highlighted in red). This is consistent with the amount of data in each table (LineItem has many attributes, while Region does not).

The data types that are part of the Select function vary, with fixed text (FT) being the most commonly used data type. This information can also be used to group, or classify queries. We discuss more about query classification in Section 2.3.

2.2 Comparison of Runtimes for Each Query

Table 3 shows the query runtimes in SQlite3 for three different machines, which were used to run the queries. The information for each machine follows the table.

Table 3: Query runtimes using three different machines. *The green highlighted rows show the queries with the lowest runtimes; the yellow are the middle runtimes; and the red are the queries with the longest runtimes.*

	Laptop						Surface						Desktop					
	min			max			min			max			min			max		
Q#	R	U	S	R	U	S	R	U	S	R	U	S	R	U	S	R	U	S
1	12.32	7.48	1.44	26.77	20.08	4.64	14.16	12.14	1.97	34.35	26.03	5.41	17.74	8.14	1.55	20.67	9.47	2.28
2	3.05	0.13	0.39	5.05	0.81	2.19	0.75	0.34	0.41	5.28	1.16	1.94	3.76	0.27	0.64	3.44	0.39	0.77
3	6.06	1.06	1.03	16.40	5.69	7.47	4.52	2.55	1.97	20.23	7.47	7.39	9.57	2.09	2.17	27.17	2.70	3.17
4	0.85	0.45	0.39	7.74	1.28	3.08	1.29	0.70	0.58	7.92	1.70	3.22	0.86	0.50	0.36	14.98	0.69	1.88
5	5.35	2.22	3.11	37.37	8.83	19.13	9.63	4.02	5.30	33.62	9.09	15.92	5.49	2.30	3.16	30.01	3.61	7.72
6	1.90	1.17	0.72	8.84	3.30	3.45	3.11	2.00	1.08	9.07	4.52	2.94	1.96	1.33	0.64	11.62	1.47	1.27
7*	32.90	2.59	4.48	53.25	10.81	23.06	39.11	7.28	15.89	63.94	13.08	30.23	38.19	4.47	10.58	43.46	6.20	12.44
8*	18.63	3.88	12.94	95.25	15.80	55.48	26.22	6.61	18.91	92.61	17.80	53.44	18.38	4.48	13.14	52.58	7.48	25.48
9*	49.25	9.36	36.61	263.82	49.30	185.64	76.20	16.44	57.30	200.49	42.73	131.69	103.33	10.16	39.73	209.02	12.47	48.84
10	2.49	1.33	1.17	13.71	4.14	7.00	4.04	1.98	2.03	17.71	5.53	6.77	2.53	1.42	1.09	22.43	1.88	2.86
11	1.37	0.56	0.27	5.34	2.08	2.31	1.56	0.95	0.61	6.47	2.19	2.30	1.40	0.64	0.45	3.64	0.81	1.00
12	1.94	1.08	0.86	9.97	3.23	3.89	2.94	1.67	1.25	11.44	4.98	3.77	1.95	1.30	0.66	10.08	1.58	1.31
13	31.84	6.86	24.55	138.39	26.94	110.61	54.30	11.89	42.20	94.17	23.38	70.00	33.84	7.34	26.48	37.82	7.39	30.13
14	2.26	1.25	1.02	12.79	3.83	6.03	3.48	1.77	1.70	12.12	3.98	3.80	2.34	1.20	1.13	13.02	1.41	1.45
15	1.91	1.22	0.69	9.48	3.63	3.50	2.82	1.70	1.13	8.74	3.84	2.67	1.91	1.34	0.58	11.4	1.30	0.80
16	0.42	0.39	0.03	4.33	1.23	0.69	0.62	0.48	0.11	2.78	1.16	0.47	0.43	0.39	0.03	1.90	0.42	0.16
17	7198.10	3001.91	4192.98	7149.12	2985.11	4157.77	>4265.97	>1792.7	>2462.5	21875.67	8967.84	12797.31	7114.00	3020.25	4084.64	>26663.5	>1614.20	>2686.20
18	1.95	1.11	0.83	1.99	1.06	0.66	N/A	N/A	N/A	7.84	4.64	2.95	1.98	1.25	0.72	N/A	N/A	N/A
19	2.79	1.38	1.42	14.07	4.36	5.53	N/A	N/A	N/A	10.26	5.13	5.11	2.99	1.56	1.41	N/A	N/A	N/A
20	11846.42	4087.48	5784.06	N/A	N/A	N/A	17156.17	6350.52	8761.5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
21	9.69	8.31	1.36	33.60	26.48	5.69	N/A	N/A	N/A	51.13	35.00	10.31	9.70	8.36	1.33	N/A	N/A	N/A
22*	1134.72	488.95	644.94	2837.16	1190.06	185.64	1658.94	745.69	910.75	2855.74	1226.39	1626.08	1070.36	478.89	589.73	1305.25	518.00	779.64

*R=run time U=user time S=sys time

Min Load (the min column for each query) was performed by closing all background processes with the power cord plugged in (to avoid power saver issues). Max load (the max column for each query) was run the same way, but with the CPU used by Prime95.exe on torture test mode. The '*' next to the query

numbers signify queries that were run by executing a different .sql file than the others. The 'N/A' were not run due to machine availability or other circumstances.

Most queries completed quickly, taking less than 180 seconds/3 minutes. The exceptions are Queries 17 and 20, both which took many hours to complete (see the cells highlighted in red). Query 22 was close, but below 3600 seconds (one hour). The '>' symbol represents queries which were not completed because they were taking a significant amount of time. The large difference in run times could signify a memory leak.

The computer characteristics are described below:

- Laptop:
  - CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.60 GHz
  - CPU Mark: 11,206 (cite: PassMark Software, 2022)
  - RAM:16.0 GB at 2667MHz
- Surface
  - CPU: Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz 2.50 GHz
  - CPU Mark: 3,259
  - RAM: 8.0Gb at 1867MHz
- Desktop
  - CPU: Intel(R) Core(TM) i3-9100F CPU @ 3.60GHz 3.60GHz
  - CPU Mark: 6,766
  - RAM: 16.0Gb at 2133MHz

Table 4 shows the laptop, surface, and desktop read/write storage.

Table 4: Laptop, surface, desktop storage

	Laptop C: 64% (141/221 GiB)		**Surface C: 61% (144/237 GiB)**		**Desktop D*: 26% (848/3259 GiB)**	
	Read (MB/s)	Write (MB/s)	Read (MB/s)	Write (MB/s)	Read (MB/s)	Write (MB/s)
Seq Q32T1	3038.5	344.6	1509.0	1015.6	160.3	152.1
4KiB Q8T8	556.8	322.9	532.9	189.8	1.829	1.569
4KiB Q32T1	424.2	297.8	247.0	186.0	1.844	1.454
4KiB Q1T1	24.78	113.0	30.98	71.60	0.555	1.443

*D drive was where the database files were, the only HDD used for storing databases. It also is where the program was kept. The C drive on that computer is an SSD, but that would have only came up if it used ram paging.

The results show that CPU speed has a significant impact on query performance. As seen in Table 4, the disk speed of the desktop is very slow compared to the SSDs of the laptop and surface; however, the queries ran with similar runtimes, and in some cases faster on the desktop machine. The high impact of CPU load can be seen with minimum load on the surface (the slowest at the TPC-H queries) being faster than the desktop and laptop at max load for all comparable queries with the exceptions of Queries 13 and 22 on the desktop being faster than min load surface.

RAM speed appears to be a bottleneck. By the CPU mark, the laptop is significantly faster. However, actual query performance on the laptop is only a bit faster, more closely matching the slight RAM speed bump of the laptop compared to the desktop. Note that the laptop CPU has more cores than the desktop one, which increases the CPU mark, but as SQLite3 only uses one CPU core, the single core performance is not as different as the CPU mark suggests. Furthermore, the amount of available RAM seems to not be a factor; while running, the RAM usage was considerably below 1.19 GB, the size of the database file

used. In the case of the min load and 16 gig machines, it is likely fastest to save the entire database to RAM for queries, instead of waiting for them to load from disk. However, given how little the disk speed seems to matter, the savings appear to be minimal (PassMark, 2022).

The data show that the speeds compare similarly for all machines (fast queries are fast on all machines, slow queries are slow, etc.), and the level of CPU limiting (seen in the difference between the min and max load for each machine) is relatively consistent among machines. This suggests that most queries are similar in the type of load they put on the computer running them, with the exception of their amount of CPU dependence. Further research can be done to determine if this makes it a good benchmark for real-world performance.

2.3 Query Classification Overview of Operators and Optimization Potential

The 22 queries of the TPC-H decision-support benchmark include optimized, unoptimized, and partly optimized queries (TPC-H.org, 2022). This diversity allows the benchmark to effectively measure the query optimization and plan selection capabilities of a given DBMS. Thus, taking these differing levels of optimization into consideration, the queries can be classified according to their number and types of joins, selections, and subqueries. Table 5 lists each of the queries along with a count of these corresponding features.

Table 5: Classification of each query via feature quantity

Query Number	Number of Selections (top level)	Inner joins	Subqueries	Inner joins within subqueries	Subqueries within subqueries	Inner joins within subqueries within subqueries	Left outer joins within subqueries
1	10	0	0	0	0	0	0
2	8	4	1	4	0	0	0
3	4	2	0	0	0	0	0
4	2	0	1	1	0	0	0
5	2	6	0	0	0	0	0
6	1	0	0	0	0	0	0
7	4	0	1	3	0	0	0
8	2	0	1	4	0	0	0
9	3	0	1	6	0	0	0
10	8	3	0	0	0	0	0
11	2	2	1	2	0	0	0
12	3	1	0	0	0	0	0
13	2	0	1	0	0	0	1
14	1	1	0	0	0	0	0
15	5	1	1	0	0	0	0
16	4	1	1	0	0	0	0
17	1	1	1	1	0	0	0
18	6	2	1	0	0	0	0
19	1	3	0	0	0	0	0
20	2	1	1	0	2	2	0
21	2	1	2	0	0	0	0
22	3	0	1	0	2	1	0

Queries 17, 20, and 22 have significantly long runtimes (see Section 2.2). In particular, Queries 20 and 22 were very long. These longer runtimes can possibly be attributed to the nesting of expensive inner joins operations within subqueries of subqueries, as described in Table 5. Based somewhat on these findings, we developed relational diagrams for Query 20 and applied some rules to optimize this query (see Section 2.4).

In comparison, Query 17 has an even longer runtime than Queries 20 and 22, but only has a single level of subqueries with inner joins. In closer examination, Query 17 performs an inner join before selection, and then uses a < operator when forming the subquery. The subquery then possesses the same inner joins, made even more inefficient by the large number of tuples in both the LineItem and Part tables

(6,000,000 and 200,000 respectively), which are two of the largest tables in the database (this can also be seen in Table 1). Because the subquery possesses the same inner joins as the primary query, the performance of Query 17 can be drastically improved by replacing the entirety of the subquery with its selection element, 0.2 * avg(l_quantity). This eliminates redundant computations while remaining consistent with the subquery's results. Query 17 can be further optimized by executing selection statements before the inner joins (as explored in Section 2.4.1 and Section 2.4.2), reducing the size of the joins computation and thereby improving the query's runtime.

Finally, Query 13 also has a runtime that is longer than most queries (except for Queries 17, 20, and 22). This is likely due to the use of a left outer joins within the query's present subquery, one of the more expensive operations that can be performed. Within the entire set of 22 TPC-H queries, Query 13 contains the only instance of an outer joins—and by extension the only measure of how a DBMS handles an outer join—making the query even more noteworthy when benchmarking.

Overall, although TPC-H's 22 queries can be classified according to Table 5, it is important to note that observing these individual elements does not yield an exact estimate for query pre-optimization. Multiple factors, such as those observed in the brief analysis of Query 17, can contribute to much longer runtimes than otherwise expected when observing the table along. However, because each of the observed elements has a significant impact on runtime, observing them in this way can assist in understanding where query planning inefficiencies may arise. This holds especially true as queries develop nested subqueries, potentially leading to greater complexities and inefficiencies as depth increases. Thus, classifying the 22 queries in this way can provide valuable surface-level insight into a given DBMS's query optimization process, plan selection, and required execution time.

2.4 Discussion of TPC-H Optimization

As discussed previously, some queries written for the TPC-H database have potential for optimization. Most queries involve overly complex operations, such as accessing multiple tables that might be correlated, and have long subqueries, which can lead to long run times.

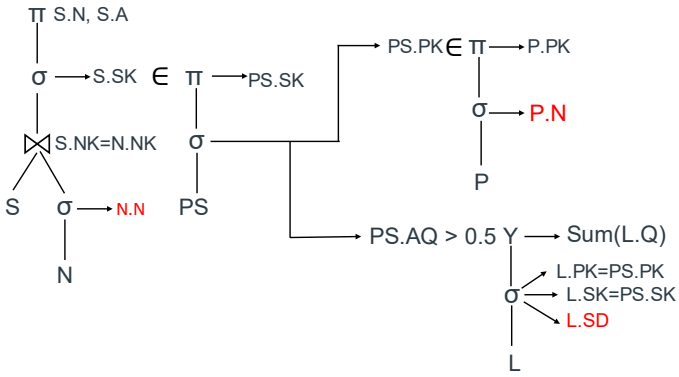
2.4.1 Literature Review

Our initial literature review of relevant research suggested that Queries 17, 20, and 22 were the 'worst performing' in terms of runtime. Our results in Table 3 are consistent with these findings. This research shows that flattening the subqueries within these queries can improve the turnaround time and performance by at least two orders of magnitude (Dreseler et al., 2020). Specifically, the subquery in Query 20 goes through and cross-checks two items and appears to subquery every single entry in another table. In addition to flattening subqueries another aspect that can be changed about the queries is the use of implicit joins such in the form of **FROM table1, table2 WHERE table1.attribute=table2.attribute,** rather than explicitly stating **table1 JOIN table2 on something...** by doing this step alone the turnaround time is improved by at least 50% on Queries 2,5,7 and 11 (Dreseler et al., 2020).

Because optimizing queries involves pushing down the cardinalities, one possible way to improve performance is to look at the statements with the most selective predicates first and have them evaluated first. This indicates that operations such as joins aggregations should be pushed down and ordered until they are in their optimal position. Research suggests that by doing this the runtimes of Queries 17 and 20 are improved twofold in the case of Query 17 and almost fivefold in the case of Query 20 and 21 (Dreseler et al., 2020). Additionally, since most queries involve searching different tables multiple times, it would be helpful to create and utilize an index. This can be done by creating indices on tables that are being searched within the queries. This has the potential to vastly improve runtime, as an index helps fetch data more quickly from tables (a process which occurs multiple times in various queries) at the cost of additional writes and storage space (Petrovic, 2018).

2.4.2 Example Optimization of Query 20

Figure 1 shows the Relational Algebra Diagram of Query 20, as it is written for the TPC-H database. All properties in red are being compared to constants.



L=LineItem
P=Part
PS=PartSupp
N=Nation
S.N=Supplier Name
S.A=Supplier Address
S.S=Supplier SuppKey
S.NK=Supplier NationKey
N.N=Nation Name
PS.SK=PartSupp SuppKey
PS.PK=Part Supp PartKey
P.PK= Part PartKey
L.Q = LineItem Quantity
L.PK=LineItem PartKey
L.SD=LineItem ShipDate
∈=Element of Symbol

Figure 1: Relational algebra diagram for TPC-H Query 20

Figure 2 shows an optimized version of Query 20.

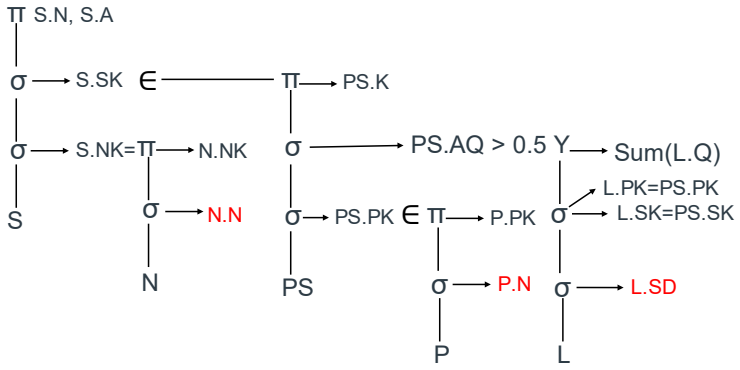


Figure 2: First draft of Query 20 with minor optimizations

The first draft of Query 20 pushes down the selections based on constants. For example, we know that only one nation has the name we are looking for, so the selection on table Nation will only get one result. Similarly, doing the checks on the part names and ship date will reduce the number of entries in the Part and LineItem tables that need to be checked further up the tree. However, it still must search the LineItem table once per part supplier entry that passes the part key check. The version shown in Figure 3 addresses this issue.

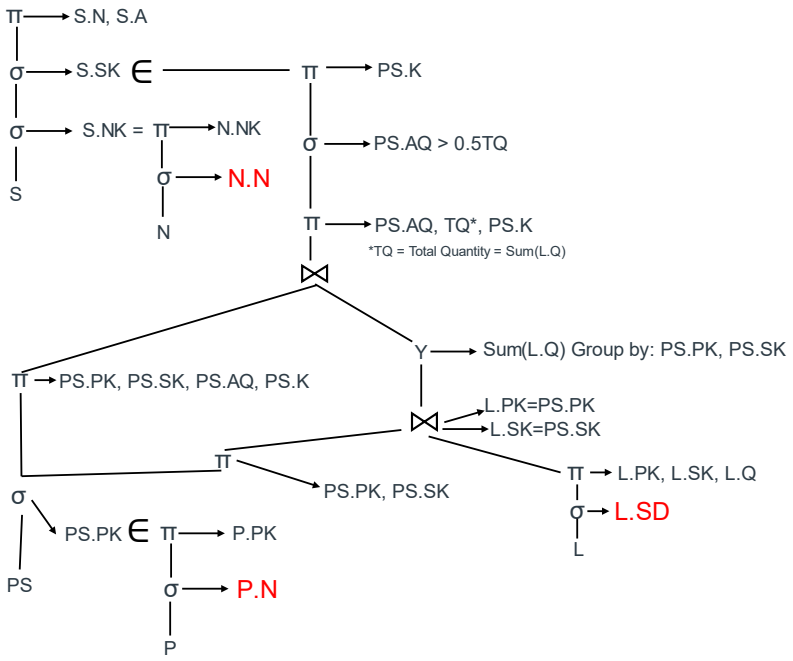


Figure 3: Second draft of Query 20 with more optimizations

This query is a major change to the plan, instead of using a sub query that runs over the LineItem table many times, it instead looks at each entry once, making a running total for each SuppKey and PartKey pair, then joins it with the rest of the PartSupp table, meaning that the selection against the total quantity does not need to do a sub query each time. The Snowflake documentation (Snowflake.com, 2022) shows that in the base table, LineItem has 6 million elements, meaning each query on it needs to examine a huge number of elements, likely causing Query 20 to be so taxing.

2.5 Identification of Correlations Between Columns

Query optimizers (i.e., logical optimization and physical optimization) typically assume some independence between columns in a relational database; however, this is not true of most real-world datasets (Ilyas et al., 2004). For example, databases can use different columns to represent correlated information (e.g., zip code vs. city name), and in the case of TPC-H, the status of an order placed and its date are correlated, even though their value domains differ (Nguyen et al., 2014). The clear correlations are involved in declared foreign key and primary key constraints, regardless of column data types. To identify additional column correlations, the individual attributes can be closer examined.

3. Conclusions and Recommendations

The TPC-H decision-making benchmark is an effective tool for the quantitative and qualitative analysis of database management systems (DBMS). The benchmark provides a baseline of easily optimized and executed queries to test these systems, alongside much harder, unoptimized query plans to identify their limits and potential pitfalls. The TPC-H benchmark supports some common organizational processes, such as comparing sales figures, predicting revenues, and evaluating the effects of making different decisions.

Our analysis of the query runtimes showed a large variation. Some queries take many hours to run, while others run in seconds. In closer examination of the characteristics of the individual queries, there are opportunities to re-write the queries to improve their efficiency. We identified ways to implement optimization techniques to improve the runtime of some of the queries, and different ways to classify the

queries based on the types of processes they support and the tables that they access. Although these queries are structured in a particular way to encourage comparison between different DBMS, they can be examined and re-written for educational purposes.

The demand for data management specialists is growing; therefore, continuing to evaluate the features of this benchmark is essential if pursuing this career path. There are many opportunities for further investigation of these queries; for example, continuing to re-write and re-structure the queries to improve their efficiency, and to compare the performance of the queries among different DBMS.

4. References

Benchant. (2022). *What is Database Benchmarking?* https://benchant.com/blog/database-benchmarking

Blue Bird Network. (2021, March 25). *How Is Data Being Used By Businesses Today?* Bluebird Network. https://bluebirdnetwork.com/how-is-data-being-used-by-businesses-today

Chaudhuri, S. (1998). *An Overview of Query Optimization in Relational Systems (paper)* (Proceedings of 1998 ACM PODS). Association for Computing Machinery, Inc. https://www.microsoft.com/en-us/research/publication/an-overview-of-query-optimization-in-relational-systems-paper/

Dreseler, M., Boissier, M., Rabl, T., & Uflacker, M. (2020). Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.*, *13*(8), 1206–1220. https://doi.org/10.14778/3389133.3389138

Ilyas, I., Markl, V., Haas, P., Brown, P., & Aboulnaga, A. (2004). *CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies.* 647–658. https://doi.org/10.1145/1007568.1007641

Nguyen, H. V., Müller, E., Andritsos, P., & Böhm, K. (2014). Detecting Correlated Columns in Relational Databases with Mixed Data Types. *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. https://doi.org/10.1145/2618243.2618251

PassMark. (2022, April 26). *PassMark Software—List of Benchmarked CPUs*. https://www.cpubenchmark.net/cpu_list.php

Petrovic, B. (2018, November 27). SQL index overview and strategy. *SQL Shack - Articles about Database Auditing, Server Performance, Data Recovery, and More*. https://www.sqlshack.com/sql-index-overview-and-strategy/

Raza, M., & Wickramasinghe, S. (2021, December 9). *DBMS: Database Management Systems Explained*. BMC Blogs. https://www.bmc.com/blogs/dbms-database-management-systems/

Snowflake.com. (2022). *Sample Data: TPC-H — Snowflake Documentation*. https://docs.snowflake.com/en/user-guide/sample-data-tpch.html

TPC. (2021). *TPC Benchmark H Standard Specification Revision 3.0.0*. Transaction Processing Performance Council (TPC).

TPC-H.org. (2022). *TPC-H Homepage*. https://www.tpc.org/tpch/