CS542 Project: TPC-H Analyzed

Phong Cao, Tyler Nardone, Olivia Raisbeck

Department of Computer Science, Worcester Polytechnic Institute

CS542: Database Management Systems

Prof. Mohammed Al-Kateb

April 23, 2024

I. Introduction

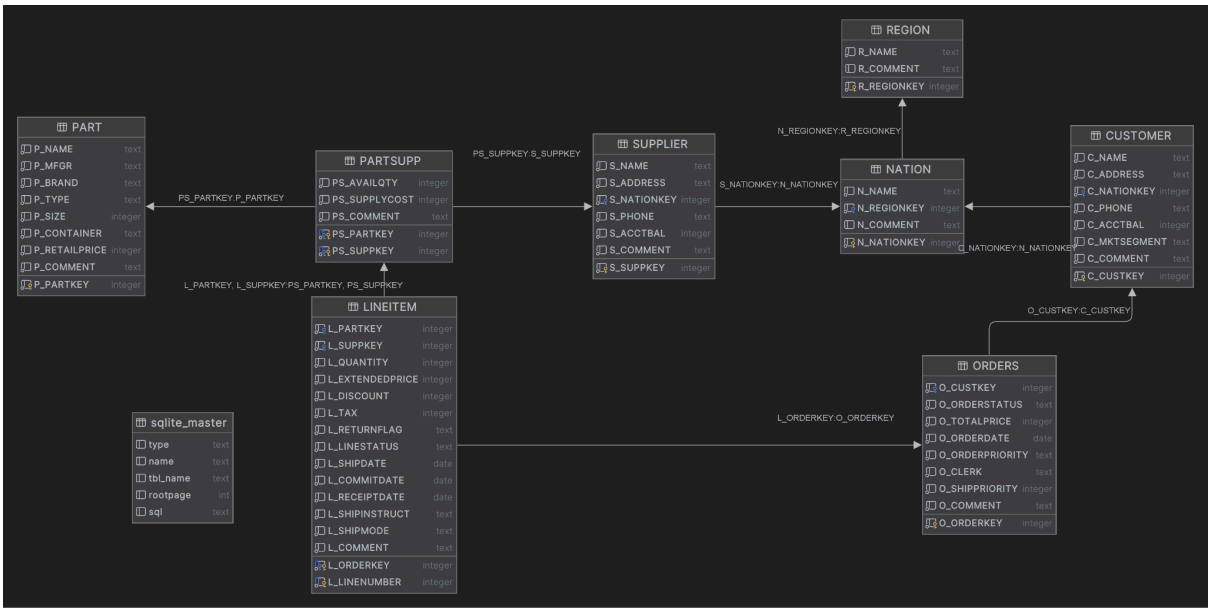
The TPC-H is a decision support benchmark. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. [1]

This project seeks to undertake a thorough and systematic evaluation of the twenty-two queries that make up this benchmark. This will include evaluating the entity-relationship nature of the underlying database, determining the output of each query, and evaluating the runtime of each query. Along with the runtime and output of each query, the textual nature of each query will be thoroughly analyzed to identify areas that have a high degree of complexity and are possibly sub-optimized.

In addition, this project will seek to examine each query by breaking it down into an equivalent relational algebra tree, and using cardinality estimations as a proxy to gauge the runtime complexity of each query. By estimating the runtime, it will be possible to test if the runtimes that are found by simply running each query align with the estimated complexity as dictated by the relational algebra and cardinality estimates.

II. Overview TPC-H

a) Entity relationship diagram:



b) Database size:

For the Customer table, we have 150,000 rows and 8 columns. In there, we have 5 text data types columns and 3 integers columns.

For the Lineitem table, we have 6,001,215 rows and 16 columns. In there, we have 8 integer data type columns, 3 date data type columns, and 5 text data type columns.

For the Nation table, we have 25 rows and 4 columns. In there, we have 2 integer columns and 2 text columns.

For the Orders table, we have 1,500,000 rows and 9 columns. In there, we have 4 integer columns, 4 text columns, and 1 date column.

For the Part table, we have 200,000 rows and 9 columns. In there, we have 3 integer columns and 6 text columns.

For the Partsupp table, we have 800,000 rows and 5 columns. In there, we have 4 integer columns and 1 text column.

For the Region table, we have 5 rows and 3 columns. In there, we have 1 integer column and 2 text columns.

For the Supplier table, we have 10,000 rows and 7 columns. In there, we have 3 integer columns and 4 text columns.

From the above information about the size of every table, it is possible to predict the running time of the query. More specifically, the Lineitem, Orders, and Partsupp tables are three tables having the biggest size (amount) of data. This will make any query working with it work through a huge amount of data and cause a slow execution time. In contrast, working with tables like region, country, and supplier which are lightweight amounts of data will make the execution time incredibly fast.

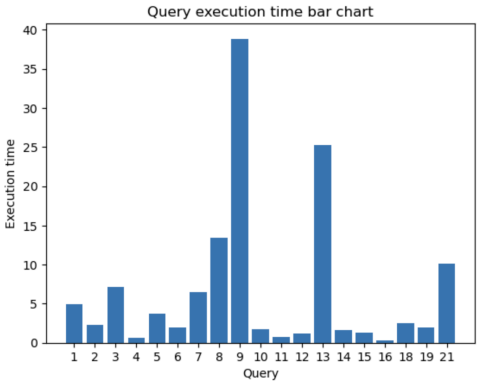
c) Execution time:

After executing 22 queries from the TPC-H database, the following results were obtained.

	Query	Execute time (s)
0	1	4.966
1	2	2.306
2	3	7.2
3	4	0.598
4	5	3.702
5	6	1.98
6	7	6.505
7	8	13.483
8	9	38.871
9	10	1.767
10	11	0.738
11	12	1.229
12	13	25.226
13	14	1.634
14	15	1.247
15	16	0.288
16	17	4933.62
17	18	2.528
18	19	1.92
19	20	6508.47
20	21	10.132
21	22	683.172

From this diagram above, it can be seen that queries 17, 20, and 22 take far more time to execute than others. The execution times for each query are 4933.62, 6508.47, and 683.172 respectively. These are tremendously longer runtimes compared with the remainder of the queries, in some cases taking longer by a factor of over 150.

The diagram below shows the runtimes excluding these three outliers.



Now, it is easier to consider and classify the execution time of all other queries. We can classify these queries into 3 other types. We consider a query slow if the execution time is above 10 seconds, a fast query if the execution time is between 2 and 10 seconds, and a speed query if the execution time is lower than 2 seconds. With those class definitions above, we have the result of classifying queries based on their execution time.

+ Speed: 4, 11, 16
+ Fast: 1, 2, 3, 5, 6, 7, 10, 12, 14, 15, 18, 19
+ Slow: 8, 9, 13, 21
+ Extremely slow: 17, 20, 22

d) Query analysis:
As we just classify the queries by their execution time, now we will go through the results and try to give some insight into the main idea of each query. To do that, for every query, we will provide the result, the size of the query's output, the insight of the query which is the business idea, and the main idea of how SQL works in this query.

A) For Speed query:

Q4)
Result:

```
   O_ORDERPRIORITY  order_count
0         1-URGENT        10569
1           2-HIGH        10385
2         3-MEDIUM        10377
3  4-NOT SPECIFIED        10440
4            5-LOW        10393
Size of query:  (5, 2)
```

Insight:
The query provides the total order in a specific time frame and orders them by order's priority.

Idea:
This query executes very fast because it mostly just works with only one table which is the orders table. The only thing that we need to merge with other tables is through the key constraint. It also just uses only one group which makes the query less complicated and increases the execution time.

Q11)
Result:

```
   PS_PARTKEY        value
0       26964  19453391.03
1      181683  17650015.20
2      159774  17150761.56
3      181556  16128801.21
4      164951  15811680.50
Size of query:  (869, 2)
```

Insight:
This query provides the sum of the total supply available cost of one country and group by the parts.

Idea:
This query has only aggregation using the sub-query. Moreover, this query mostly works with just integer variables and calculates them then groups them into groups of parts.

Q16)
Result:

```
    P_BRAND                 P_TYPE  P_SIZE  supplier_cnt
0  Brand#41      MEDIUM BRUSHED TIN       3            28
1  Brand#11   STANDARD POLISHED TIN      39            24
2  Brand#13   MEDIUM BRUSHED NICKEL       1            24
3  Brand#21  MEDIUM ANODIZED COPPER       3            24
4  Brand#22    SMALL BRUSHED NICKEL       3            24
Size of query:  (18263, 4)
```

Insight:
Query 16 gives us an idea of how many suppliers can have enough qualifications for the attributes of supply parts.

Idea: The idea of query 16 is to use conditions to choose only those who meet the requirements and group those information by the brands.

In conclusion, from these speed queries' insights, it is a perfect combination of using aggregation, grouped by conditions. Most of the conditions already reduce the amount of data the aggregation and grouped by have to deal with. Also although these queries use a huge amount of data they just have one large table only and they don't really merge with another large table. Moreover, the business intelligence between these queries is really useful which makes these speed queries helpful in business and gives good insights. In conclusion, we can say that these queries are the most optimized queries in TPC-H.

B) For Fast query:

Q1)

Result:

```
  L_RETURNFLAG L_LINESTATUS   sum_qty  sum_base_price  sum_disc_price  \
0            A            F  37734107    5.658655e+10    5.375826e+10
1            N            F    991417    1.487505e+09    1.413082e+09
2            N            O  76631223    1.149317e+11    1.091863e+11
3            R            F  37719753    5.656804e+10    5.374129e+10

     sum_charge    avg_qty     avg_price  avg_disc  count_order
0  5.590907e+10  25.522006  38273.129735  0.049985      1478493
1  1.469649e+09  25.516472  38284.467761  0.050093        38854
2  1.135576e+11  25.502088  38248.108587  0.050000      3004900
3  5.588962e+10  25.505794  38250.854626  0.050009      1478870
Size of query:  (4, 10)
```

Insight:

Query 1 provides a summary of all line items shipped in a specific amount of time, which is in three days.

Idea:

The idea of query 1 is to use lots of aggregation in just one table. So for query 1, the reason for its fast execution time is that it mostly works with numbers and aggregation only. It does not use or merge with another table, which means it just calculates on its own.

Q2)

Result:

```
   S_ACCTBAL              S_NAME          N_NAME  P_PARTKEY          P_MFGR  \
0    9938.53  Supplier#000005359  UNITED KINGDOM     185358  Manufacturer#4
1    9937.84  Supplier#000005969         ROMANIA     108438  Manufacturer#1
2    9936.22  Supplier#000005250  UNITED KINGDOM        249  Manufacturer#4
3    9923.77  Supplier#000002324         GERMANY      29821  Manufacturer#4
4    9871.22  Supplier#000006373         GERMANY      43868  Manufacturer#5

                          S_ADDRESS          S_PHONE  \
0          QKuHYh,vZGiwu2FWEJoLDx04  33-429-790-6131
1  ANDENSOSmk,miq23Xfb5RWt6dvUcvt6Qa  29-520-692-3537
2              B3rqp0xbSEim4Mpy2RH J  33-320-228-2957
3                      y3OD9UywSTOk  17-779-299-1839
4                         J8fcXWsTqM  17-813-485-8637

                                       S_COMMENT
0                     uriously regular requests hag
1  efully express instructions. regular requests ...
2  etect about the furiously final accounts. slyl...
3  ackages boost blithely. blithely regular depos...
4  etect blithely bold asymptotes. fluffily ironi...
Size of query:  (460, 8)
```

Insight:

The idea of query 2 is to give all the information of the supplier and order it with the supplier who can provide it with the minimum cost.

Idea:

The main idea of query 2 is to merge different tables together through keys and conditions. Through that, we project all the columns we need for the information of the supplier. After all, we order to find out who provides the minimum cost.

Q3)
Result:

```
   L_ORDERKEY      revenue O_ORDERDATE  O_SHIPPRIORITY
0     5660420  389343.1796  1995-03-08               0
1     2118374  384042.9549  1995-03-19               0
2     3837441  382568.0383  1995-02-14               0
3      564738  378219.2915  1995-03-06               0
4     5006400  377083.4322  1995-03-18               0
Size of query:  (11264, 4)
```

Insight:
Query 3 provides the shipping priority based on the revenue it could make in a specific time frame.

Idea:
Same idea as the above queries, query 3 also merges customer, orders, and lineitem tables together by keys and then gives the time frame through the conditions, after all, it uses 3 groups and orders it to know which of them are profitable.

Q5)
Result:

```
            N_NAME       revenue
0         ROMANIA  5.483336e+07
1          RUSSIA  5.381822e+07
2         GERMANY  5.183593e+07
3          FRANCE  5.180258e+07
4  UNITED KINGDOM  4.910800e+07
Size of query:  (5, 2)
```

Insight:
Query 5 provides us with the total revenue earned from Lineitem transactions in countries and orders it by descending.

Idea:
The main idea of query 5 is mostly similar to query 3, in which we also try to merge all the tables by their constraints and keys. After that, it has some conditions to filter out the time frame they want and then uses group-by and order-by “country name” in descending order.

Q6)
Result:

```
        revenue
0  1.031663e+08
Size of query:  (1, 1)
```

Insight:
Query 6 provides us with the total number of items shipped in a year.

Idea:
The idea of query 6 is pretty basic, which is just working on one Lineitem table applying different conditions to get the time frame we want, and then making a sum aggregation function. Moreover, most of the data this query goes through is an integer which makes the execution time faster than lots of other fast queries.

Q7)
Result:

```
  supp_nation cust_nation  l_year       revenue
0        IRAQ        PERU    1995  5.659699e+07
1        IRAQ        PERU    1996  5.798743e+07
2        PERU        IRAQ    1995  5.141655e+07
3        PERU        IRAQ    1996  5.251922e+07
Size of query:  (4, 4)
```

Insight:
Query 7 determines the value of goods shipped between two nations

Idea:
The idea of query 7 is to get data from the sub-query. In that sub-query, we merge multiple tables by their constraints and keys, and then we give conditions to get the time frame and other conditions to choose the countries we want to research. After returning to the main query, we use sum aggregation to calculate the total revenue.

Q10)
Result:

```
   C_CUSTKEY             C_NAME      revenue  C_ACCTBAL         N_NAME  \
0      29518  Customer#000029518  695992.4969    9244.62        GERMANY
1      39412  Customer#000039412  691788.4046    8888.43           PERU
2      54826  Customer#000054826  668052.7708    1018.83          CHINA
3      59413  Customer#000059413  651289.9559    9948.30         CANADA
4     127486  Customer#000127486  648241.9962    6630.53  UNITED STATES

                          C_ADDRESS          C_PHONE  \
0          9l0hP7jIItXcPieiSIaKsQfiCn  17-567-804-8028
1              FYhhcuRk4eRpF,hylCoAP  27-684-509-6399
2  0qQN5BZXBDTUmndXQP,LO00e9ptdAWT3vMD  28-849-358-2535
3     YfRCT3dnnD3rwefu9r F1Rgl3a4nV4 x  13-419-750-2952
4                    4Cj9IM RXBb9JvRV  34-842-719-7704

                                       C_COMMENT
0                   wake. blithely ironic packages
1             l packages. special pinto beans are c
2  riously above the final packages. pinto beans ...
3  nic, bold packages among the furiously regular...
4  ly ironic foxes. pending, ironic pinto beans d...
Size of query:  (37302, 8)
```

Insight:
Query 10 gives us the top customers listed in descending orders of lost revenue.

Idea:
Query 10 is also one of the basic queries. For this query, we connect customer, orders, lineitem, and nation tables by their constraints and keys. Then we also use conditions to select the timeframe and return. After that, we use one aggregation only to calculate the sum of lost revenue and then display the columns in the customer to show the customer's information.

Q12)
Result:

```
  L_SHIPMODE  high_line_count  low_line_count
0        AIR             6293            9123
1       RAIL             6313            9320
Size of query:  (2, 3)
```

Insight:
This query provides us with the shipping modes and the order priority counts.

Idea:
The special nature of query 12 to other queries in TPC-H is that we use case conditions in this query which will help us to split into high and low line counts. However, we also just work only with 2 tables for

this query which are the orders and the lineitem. The rest of the query has the same idea as most of the query above is uses conditions to select the timeframe. After all, the main point is just using sum aggregation to calculate. This query is fast because however 2 tables we are working with are big, and the things we are working on the whole time are in integer data type.

Q14)
Result:

```
   promo_revenue
0      16.417037
Size of query:  (1, 1)
```

Insight:
Query 14 provides us with the market response to a promotion in a special campaign.

Idea:
Like query 12 above, the idea of query 14 is also basic. It just uses the case function and after that is just conditions to select the time frame. This query is able to run fast because it is working with only 2 tables which are Lineitem and Part, which are merged by the constraint keys. After all, we also have 1 aggregation only which is the sum to calculate the total.

Q15)
Result:

```
   S_SUPPKEY             S_NAME                   S_ADDRESS          S_PHONE  \
0       2932  Supplier#000002932  gAIrgSCdvtJltNKuZKRGYeYLRf  14-722-108-2914

   total_revenue
0   1.926296e+06
Size of query:  (1, 5)
```

Insight:
Query 15 gives us the maximum total revenue0 of one supplier.

Idea:
The idea of how this query works is firstly to calculate the sub-query which is the revenue0 first. This revenue0 execution time should also be fast because it just uses 1 aggregation from 1 table online which is lineitem. After that, to the main query, we apply conditions to find the maximum revenue0 of the supplier. This query runs fast also because it works with mostly the integers.

Q18)
Result:

```
               C_NAME  C_CUSTKEY  O_ORDERKEY O_ORDERDATE  O_TOTALPRICE  \
0  Customer#000128120     128120     4722021  1994-04-07     544089.09
1  Customer#000144617     144617     3043270  1997-02-12     530604.44
2  Customer#000066790      66790     2199712  1996-09-30     515531.82
3  Customer#000015619      15619     3767271  1996-08-07     480083.96
4  Customer#000147197     147197     1263015  1997-02-02     467149.67

   sum(l_quantity)
0              323
1              317
2              327
3              318
4              320
Size of query:  (9, 6)
```

Insight:
Query 18 provides us the information about the customer and the order which is considered to be a large quantity order.

Idea:
Query 18 is also a basic query. This is because it merges customer, orders, and lineitem together by their key constraints. Then through that, we find the order key we want and then just need to group by and provide the information of the customer and order.

Q19)
Result:

```
        revenue
0  3.039604e+06
Size of query:  (1, 1)
```

Insight:
Query 19 reports the gross discounted revenue for all orders by three types of delivery which are by air, and delivered by person.

Idea:
The main idea of this query is to wrap the query with one aggregation which is the sum. The whole other part of this query is just the “or” function and inside there are conditions. This means we have three independent sets of conditions. This is also a really basic query and it is considered as a fast query because it just works with two tables which are lineitem and part.

Overall, for these fast queries, mostly all of the ideas are basic and easy to understand. This is because fast queries tend to have a straightforward code. One more factor that makes these considered fast queries because, most of the time, they work with a lot of integers instead of text data type. Because of this, even if there are some queries that work with a huge amount of data merged together, it still is fast because the query already uses an aggregation function to deal with that. Another factor is that we do not use group by function which is the data that have too many values. All of the groups by function are applied to small data like country or customer after applying conditions. One more thing is the sub-query factor. All of the fast queries do not have complex and repeatedly sub-query structures.

C) For Slow query:
Q8)
Results:

```
   o_year  mkt_share
0    1995   0.035234
1    1996   0.043039
Size of query:  (2, 2)
```

Insight:
The idea of query 8 is to determine how the market share of a nation or a region has changed over a timeframe.

Idea:
First, we get all nations from the sub-query which gets the date, calculates the volume, and gets the nation from part, supplier, lineitem, orders, customer, nation n1, nation n2, and region. These tables are merged by their constraint keys in conditions and then there are also other conditions to filter out the date, nation key, and p_type we want. We can see here that one of the reasons this query is considered a slow query is because this query has more than 1 time running all through the table when working with the nation. After all, we just apply the sum aggregations and group by the order year. However, here the query slows down one more time because we are using 2 sum aggregation for the mkt_share calculation.

Q9)
Result:

	nation	o_year	sum_profit
0	ALGERIA	1998	2.901212e+07
1	ALGERIA	1997	4.943342e+07
2	ALGERIA	1996	4.560312e+07
3	ALGERIA	1995	4.631674e+07
4	ALGERIA	1994	4.859656e+07

Size of query: (175, 3)

Insight:
Query 9 determines how much profit is made on a given line of parts, broken out by supplier nation and year.

Idea:
Query 9 has the same idea as query 8 above, which is also getting information from the sub-query. In there, we take nation, o_year, and amount information through merge part, supplier, lineitem, partsupp, orders, and nation by their constraint keys in the condition. After that, we apply one sum aggregation and group it by nation and o_year and order it by nation and o_year in descending order.

Q13)
Result:

	c_count	custdist
0	0	50004
1	10	6595
2	9	6595
3	11	6032
4	8	5923

Size of query: (42, 2)

Insight:
This query gives us the relationship between the customers and their size of orders. This means this query gets the distribution of customers by the number of orders they have made and counts how many customers have no orders.

Idea:
Similar to the two queries above, first we get the data through a sub-query. Here we left-join Customer with the Orders table. However, this query is slowed down further because now it has to work with text data type which is the o_comment. The difference of using the sub-query in query 13 and then other two slowed queries is it has the aggregation and group-by within the sub-query. After that, we project the c_count and count custdist and then group it by c_count and order it by custdist and c_count descending.

Q21)
Result:

	S_NAME	numwait
0	Supplier#000002340	25
1	Supplier#000004318	21
2	Supplier#000005601	19
3	Supplier#000009177	18
4	Supplier#000004804	17

Size of query: (415, 2)

Insight:
Query 21 gives us an insight into the total numwait amount of each supplier.

Idea:
The idea of this query is quite complex. In this query, we have two sub-query that serve as an “exist” condition for the main query. However, these two sub-query sizes are much smaller than other

sub-queries in other slow queries above. We can call these mini sub-queries because they just have one table in both of the sub-queries which is the Lineitem table. After that, the query merges four tables which are Supplier, Lineitem, Orders, and Nation together by their key constraints. This query is considered a slow-running query also because it has to execute the Lineitem multiple times while these serve as the query's conditions. After all, it has one count aggregation for the numwait and groups all the information by supplier name.

In conclusion, we can conclude that there are similarities between these slow queries. First is the query's structure. Unlike the fast queries above where the code tends to have a straightforward and simpler structure, the slow queries use a lot of sub-query serving as their conditions. This may cause slow speed in the queries because we have to run many times in the same table. As we can see, the sub-query in query 21 is much smaller than other sub-query in other slow queries, and it gives query 21 a much faster execution time than other slow queries where they use a huge amount of sub-query and have to execution their multiple times as conditions. The second similarity between the slow queries is they have to deal with a huge amount of data. All of these queries above have to merge multiple tables by the key constraints. This should be an important factor in the execution time of the query because the table itself already has a hundred thousand rows before and when merged with other tables it will increase exponentially.

D) For Extremely slow query:
Q17)
Result:

```
      avg_yearly
0  344045.272857
Size of query:  (1, 1)
```

Insight:
Query 17 determines how much average yearly revenue would be lost if orders were no longer filled for small quantities of certain parts. This gives us insight about orders of small quantities.

Idea:
The idea of this query is first to use a sub-query as the conditions. In this sub-query, we connect two tables which are Lineitem and Part. Then we have one aggregation which is averaged to calculate the amount of quantity we should have in the condition. This can be one of the important factors for the extremely slow execution time of query 17. The sub-query though is not complicated but we have to calculate and keep doing it again and again with the huge amount of data that merges the Lineitem and Part tables together. Also, we already know that the Lineitem table has more than six million rows and is the largest table in our database. Because of that even though we use average aggregation already, it is still a huge amount of data for the calculation. In the main query, we also have 2 more conditions both are text data and these also search through the whole Lineitem merge with part tables. This is another huge amount of data we have to process and this time we even need to work with the text data type which is much slower than when processing with the integer data type. In conclusion, for this query, we have to rerun and reprocess the merge between the merge of lineitem and part tables multiple times so we have to deal with a huge amount of data.

Q20)
Result:

```
                S_NAME                                S_ADDRESS
0  Supplier#000000035         QymmGXxjVVQ5OuABCXVVsu,4eF gU0Qc6
1  Supplier#000000068  Ue6N50wH2CwE4PPgTGLmat,ibGYYlDoOb3xQwtgb
2  Supplier#000000080       cJ2MHSEJ13rIL2Wj3D5i6hRo30,ZiNUXhqn
3  Supplier#000000100               rIlN li8zvW22l2slbcx ECP4fL
4  Supplier#000000274                    usxbl9KSW41DTE6FAglxHU
Size of query:  (183, 2)
```

Insight:
This query gives us the suppliers who have an excess which is defined to be more than half of the parts like the given part that the supplier shipped in a given year for a given nation.

Idea:
The query has a strange structure where we can image the query in the query and in the query. This means there a query is a sub-query and serves as the condition for another query. But this query is also the sub-query for other queries and serves as a condition for the main query. This makes us have to reprocess many times and costs us a huge amount of time in the execution of the query. First, we merge Supplier and Nation together by their constraint keys. Then for each row of this merged table, we need to process through the Partsupp table which is also a large table. Moreover, for each row of the Partsupp, we have to process the Part table which is also large. This makes the amount of work increase exponentially. By basic calculation, we can know the total rows this query works through is the size of three large tables multiplied together and it's a huge number of data we have to process and work with.

Q22)
Result:

```
   cntrycode  numcust  totacctbal
0         21      955  7235832.66
1         24      920  6866012.75
2         28      907  6700667.29
3         29      948  7158866.63
Size of query:  (4, 3)
```

Insight:
Query 22 gives us the total account balance of all customers in the same countries.
Idea:
Like query 21 above, query 22 also has the query-in-query-in-query structure. However this time the first sub-query doesn't serve as a condition and it is just a normal sub-query. Because of that the execution time in query 22 is ten times faster than query 21. However, it still has a slow execution time. Also different from query 21, most of the data types query 22 has to deal with is an integer data type which makes it reasonable to have faster execution time than query 21.

In conclusion, we can see the extremely slow queries have the most complex query structure in all of the TPC-H database. For example, query 21 has the query-in-query-in-query structure. Making sub-query a condition is the main reason for the extremely slow execution time. This makes our query have to reprocess the tables a huge amount of time. Additionally, we can see the same idea that causes the extremely slow execution time is because we also use the Lineitem or other large tables as the sub-query, which is the one we have to keep reprocessing multiple times. While the amount of work in slow queries is already a lot, the amount of extremely slow queries increases exponentially. This makes these queries sub-optimal in business intelligence because it costs too much to render, and execution will cause inefficiency in a business model.

e) Relational algebra
Evaluating each query and empirically examining their structure and nature provided a good basis to hypothesize why some queries are able to execute faster than others. In some cases, it is clear to see how a given query would take very long to execute, simply by examining how the query is written.

As a secondary goal to this project, the next step was to dive deeper into the inner workings of each query by attempting to break them down into an equivalent relational algebra tree representation, to hopefully provide further insights that explain the slow runtimes of certain queries.

This process involved examining each query, and/or its component subqueries, and applying the rules that have been covered in this course and the textbook to create a tree representation that is made up of selections, projections, group-by objects, joins, etc. The relational algebra tree structure of each of the twenty-two queries can be found as an attachment in the Appendix.

The purpose of this exercise was two-fold. First, this was an effective way to provide a deeper understanding of each query and the inner workings of how a result is arrived at. In addition, the relational algebra trees provided a framework from which cardinality estimations could be applied, so as

to provide a quantitative estimate for the runtime complexity, i.e. the number of tuples, that are involved in evaluating each query.

Applying cardinality estimates allowed the following hypothesis to be tested: The queries that were identified at the beginning of this paper as being exceedingly long would produce cardinality estimates that were similarly large in proportion to the other queries.

However, in reality, this evaluation produced mixed results, and it was not clear that the queries that had the longest runtimes also shared the largest cardinality estimations based on their relational algebra trees.

Query	Runtime	Cardinality Estimate	Tuples/Second
Q8	38.871	2,667	68
Q13	25.226	1,500,000	59,462
Q17	4933.62	6,864,348	1,391
Q20	6508.473	10,216,511	1,570
Q21	10.132	80,016	7,897
Q22	683.172	3,133,334	4,586

If the longer runtimes could be explained alone by the cardinality estimates, one would expect a constant number of tuples/second. However, based on the estimates that have been produced, there is a very large degree of variance among the estimated tuple processing rate.

Further, there are other, much simpler queries, such as query 6 that consider a number of tuples similar in order of magnitude as the largest of those listed in the table above, however, this and other queries like it demonstrate significantly faster tuple processing speeds.

To this end, while the relational algebra trees and cardinality estimates have been a useful exercise in gaining a deeper understanding of the underlying structure of the queries, they are not sufficient to explain why these three queries take so much longer to process relative to the others. The true reason must lie in other factors that are beyond the scope of this project.

III. Insight

a) Hidden characteristics

- P_RETAILPRICE goes in over 900-1900 or 1100-2100 and goes up by 1.001 each row
- O_SHIPPRIORITY is 0 for all 600,000 values in the orders table
- lineitem by far the biggest table with 6,000,000+ values

b) Correlations between columns within the same table and across tables

- In supplier, S_Name is just “Supplier#0000” + s_suppkey
- In customer, C_Name is just “Customer#0000” + c_custkey
- FOREIGN KEY (N_REGIONKEY) REFERENCES REGION(R_REGIONKEY)
- FOREIGN KEY (S_NATIONKEY) REFERENCES NATION(N_NATIONKEY)
- FOREIGN KEY (C_NATIONKEY) REFERENCES NATION(N_NATIONKEY)
- FOREIGN KEY (PS_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY),
- FOREIGN KEY (PS_PARTKEY) REFERENCES PART(P_PARTKEY)
- FOREIGN KEY (L_PARTKEY, L_SUPPKEY) REFERENCES PARTSUPP(PS_PARTKEY, PS_SUPPKEY)
- FOREIGN KEY (O_CUSTKEY) REFERENCES CUSTOMER(C_CUSTKEY)
- FOREIGN KEY (L_ORDERKEY) REFERENCES ORDERS(O_ORDERKEY),

c) Factors in query that affect the execution time

i) Dimensional of a query

This seems to be the most important factor affecting the execution time of each query. This is even more so when working with a huge database that includes large volumes of data. For instance, the importance of dimensions in the database shows its importance in the extremely slow queries above, especially query 20. In those circumstances, we can see that there is a heavy and complex structure where we have a query as a condition of another query and inside another query's condition. These high dimensional structures make the query have to go back and forth to search for just one data for the whole table. That is the reason why the heavier the table we have in that sub-query, the more workload a query has to do. This means we have to reduce the dimensionality of these queries if we want to use them with a huge database. This is the most important thing to do, especially if we want to apply these queries in business, where profit is made from the query.

ii) Structure of the query

The structure is another important factor when designing a query. We can see from those ideas of queries above. The fast and speedy queries tend to have a simpler and more straightforward structure than the slow and extremely slow queries. This is because we just need to scan each table once for all the results we want. It makes the queries work more efficiently and will be more helpful in a business model.

iii) Aggregation

The third factor is the aggregation factor. We can see that fast and speedy queries tend to have just one aggregation function. However, we think we still need to experience a much larger database to conclude the importance of this factor.

iv) Order by

The fourth factor is the order-by factor. This factor also contributes to the execution time of the query. The reason for this is the more group-by the function we use, the more we create high dimensional data, and working with high dimensional data tends to have slower running or execution time.

iv) Size of database

The size of the database is one of the most important factors in the dimensionality of the data. The reason for this conclusion is obvious because the more data we have to deal with, the more workload the query has to process and deal with. This factor is already proven from the above query. We can see that the slow queries tend to work with the Lineitem table, which is the heaviest table in our database. Moreover, we can see that query 20, the slowest query has run through this table multiple times in the condition and this makes the execution time of query 20 approximately 6500 seconds, which makes for inefficient use in the business model.

v) Data type

The data type is also an important factor in the database. This is because storing a number always costs less than the text data type. Additionally, processing a number is also much easier than the text type. This is also proven when we classify the queries. The slow queries always need to work with a lot of text data types in the conditions.

vi) Results

From this benchmark, it appears that the result of the query is not a factor that affects the execution time of a query. This just affects the rendering when the query finishes and gives a result.

d) Optimization

There are multiple ways to optimize the queries above. After drawing out the relational algebra of all the queries, we notice that most of the queries have the projections after all. This means to optimize the query, we can push down these projections before merging tables. This will not affect the cardinality, however, it will reduce the workload and the memory we use. The second way is to use the selections before we merge tables. This will reduce the query's workload. Additionally, there is one more way to reduce the query's dimension. This will help us to solve the high dimension query like query 20. The way to reduce the dimension of these queries is to merge tables and not put sub-query in condition.

IV. Conclusion

This project has provided a thorough and deep look into the TPC-H benchmark and underlying database. The activities undertaken by this project have provided many insights into the structure of the entity-relationship nature of the relations that represent the data, insights into the purpose and desired output of each of the twenty-two queries, the runtime required to execute each query, the factors that affect to the execution time of the query and way to optimize it, and the abstractions produced by converting the queries into relational algebra tree structures.

By thoroughly examining each query and looking deeply into the structure of how they are written, it was possible to gain a logical understanding of why some queries would be more complex than others. Much can be learned by looking at the operations involved in component sub-queries and the size and number of times a table must be considered in a subquery. In addition, this practice provides clues as to where future optimization opportunities could take place.

As has been discussed, a secondary goal of this project was to try and explain why a small number of the queries took orders of magnitude longer than others to execute. It was hypothesized that this could perhaps be explained by the cardinality estimates or number of tuples involved in executing each query. However, as the analysis has shown, this alone does not explain the longer runtimes, since other faster-running queries also share cardinality estimates that are not far off from the long-running queries. The factor that contributes to the extremely slow queries is the high dimensional of the queries while these queries have to reprocess the tables.

Future work in this subject could include further hypothesizing why these queries take so long, and developing other quantitative tests to try and explain this variation.

However with the work that has been completed here, much has been learned about the TPC-H benchmark, and in particular how it is relevant to real-world, critical business questions that it addresses.

References

Sample queries, *deistersoftware*
https://docs.deistercloud.com/content/Databases.30/TPCH%20Benchmark.90/Sample%20querys.20.xml?embedded=true#ca2363a57661fe6bcf4df674f83d0288

Appendix

1. Query Outputs and Runtimes
2. Relational Algebra Trees
3. Cardinality Estimation Calculations
4. Cardinality Estimation Summary
5. Presentation Slides

PhongCao_QueryResult

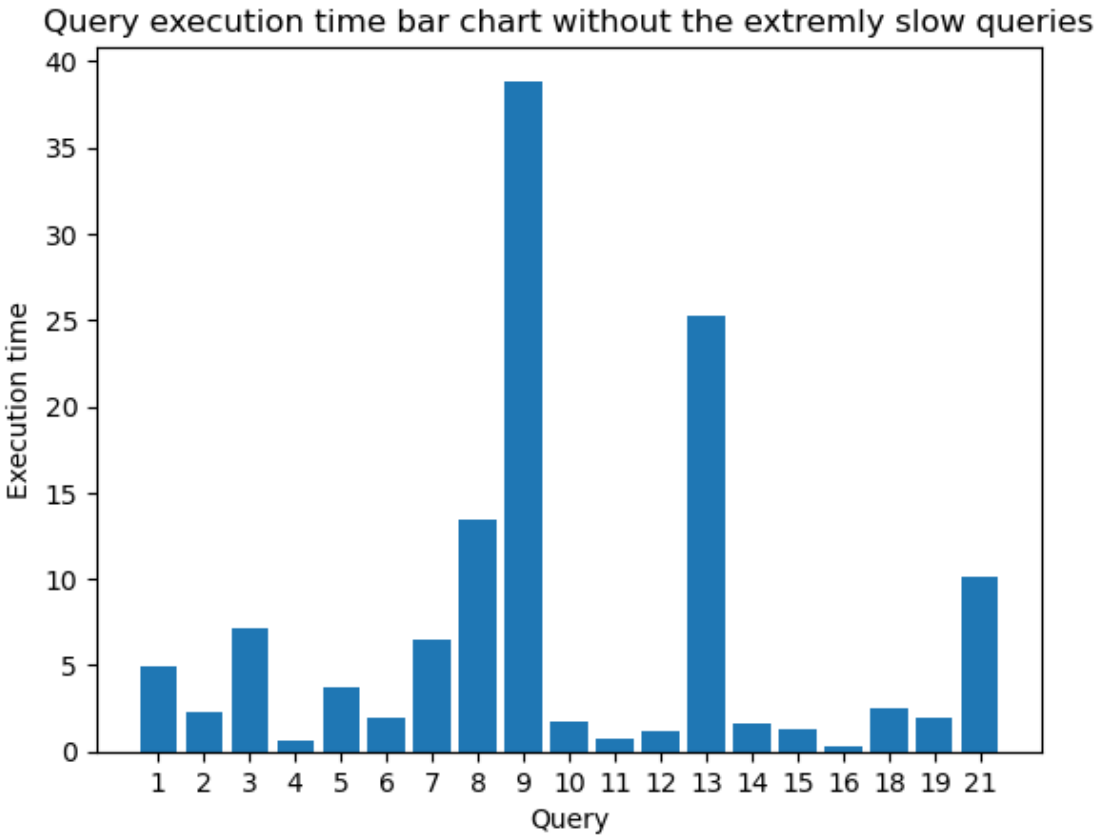
April 23, 2024

[24]:
```
pip install prettytable
```

```
Collecting prettytable
  Obtaining dependency information for prettytable from https://files.pythonhost
ed.org/packages/3d/c4/a32f4bf44faf95accbbd5d7864ddef9e289749a8efbc3adaad4a467177
9a/prettytable-3.10.0-py3-none-any.whl.metadata
  Downloading prettytable-3.10.0-py3-none-any.whl.metadata (30 kB)
Requirement already satisfied: wcwidth in c:\users\caoth\anaconda3\lib\site-
packages (from prettytable) (0.2.5)
Downloading prettytable-3.10.0-py3-none-any.whl (28 kB)
Installing collected packages: prettytable
Successfully installed prettytable-3.10.0
Note: you may need to restart the kernel to use updated packages.
```

[6]:
```
import pandas as pd
import matplotlib.pyplot as plt
from tabulate import tabulate

df = pd.read_csv("runtime.csv")
df_mid = df[(df["execution"] < 600)]

x = df_mid["QueryID"].astype("string")
y = df_mid["execution"]



def addlabels(x,y):
    for i in range(len(x)):
        plt.text(i, y[i], y[i], ha = 'center')

tick_label =␣
 ↪["1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","18","19","21"]
plt.bar(x,y, tick_label = tick_label)
plt.xlabel("Query")
plt.ylabel("Execution time")
# addlabels(x, y)
plt.title("Query execution time bar chart without the extremly slow queries")
plt.show
```

[6]: <function matplotlib.pyplot.show(close=None, block=None)>



[7]:
```
from tabulate import tabulate
head = ["Query", "Execute time (s)"]
print(tabulate(df, headers=head, tablefmt="grid"))
```

```
+----+---------+--------------------+
|    |   Query |   Execute time (s) |
+====+=========+====================+
|  0 |       1 |              4.966 |
+----+---------+--------------------+
|  1 |       2 |              2.306 |
+----+---------+--------------------+
|  2 |       3 |              7.2   |
+----+---------+--------------------+
|  3 |       4 |              0.598 |
+----+---------+--------------------+
|  4 |       5 |              3.702 |
+----+---------+--------------------+
|  5 |       6 |              1.98  |
+----+---------+--------------------+
```

```
|  6 |       7 |              6.505 |
+----+---------+--------------------+
|  7 |       8 |             13.483 |
+----+---------+--------------------+
|  8 |       9 |             38.871 |
+----+---------+--------------------+
|  9 |      10 |              1.767 |
+----+---------+--------------------+
| 10 |      11 |              0.738 |
+----+---------+--------------------+
| 11 |      12 |              1.229 |
+----+---------+--------------------+
| 12 |      13 |             25.226 |
+----+---------+--------------------+
| 13 |      14 |              1.634 |
+----+---------+--------------------+
| 14 |      15 |              1.247 |
+----+---------+--------------------+
| 15 |      16 |              0.288 |
+----+---------+--------------------+
| 16 |      17 |           4933.62  |
+----+---------+--------------------+
| 17 |      18 |              2.528 |
+----+---------+--------------------+
| 18 |      19 |              1.92  |
+----+---------+--------------------+
| 19 |      20 |           6508.47  |
+----+---------+--------------------+
| 20 |      21 |             10.132 |
+----+---------+--------------------+
| 21 |      22 |            683.172 |
+----+---------+--------------------+
```

[8]:
```
df_low = df[(df["execution"] < 1)]
print(df_low["QueryID"])

df_mid = df[(df["execution"] < 10) & (df["execution"] > 1)]
print(df_mid["QueryID"])

df_slow = df[(df["execution"] < 600) & (df["execution"] > 10)]
print(df_slow["QueryID"])
```

```
3      4
10    11
15    16
Name: QueryID, dtype: int64
0      1
1      2
```

```
2      3
4      5
5      6
6      7
9     10
11    12
13    14
14    15
17    18
18    19
Name: QueryID, dtype: int64
7      8
8      9
12    13
20    21
Name: QueryID, dtype: int64
```

[4]:
```
q1 = pd.read_csv("Q1.csv")
print(q1.head())
print("Size of query: ", q1.shape)
```

```
  L_RETURNFLAG L_LINESTATUS   sum_qty  sum_base_price  sum_disc_price  \
0            A            F  37734107    5.658655e+10    5.375826e+10
1            N            F    991417    1.487505e+09    1.413082e+09
2            N            O  76631223    1.149317e+11    1.091863e+11
3            R            F  37719753    5.656804e+10    5.374129e+10

     sum_charge    avg_qty     avg_price  avg_disc  count_order
0  5.590907e+10  25.522006  38273.129735  0.049985      1478493
1  1.469649e+09  25.516472  38284.467761  0.050093        38854
2  1.135576e+11  25.502088  38248.108587  0.050000      3004900
3  5.588962e+10  25.505794  38250.854626  0.050009      1478870
Size of query:  (4, 10)
```

[10]:
```
q2 = pd.read_csv("Q2.csv")
print(q2.head())
print("Size of query: ", q2.shape)
```

```
   S_ACCTBAL              S_NAME          N_NAME  P_PARTKEY          P_MFGR  \
0    9938.53  Supplier#000005359  UNITED KINGDOM     185358  Manufacturer#4
1    9937.84  Supplier#000005969         ROMANIA     108438  Manufacturer#1
2    9936.22  Supplier#000005250  UNITED KINGDOM        249  Manufacturer#4
3    9923.77  Supplier#000002324         GERMANY      29821  Manufacturer#4
4    9871.22  Supplier#000006373         GERMANY      43868  Manufacturer#5

                          S_ADDRESS          S_PHONE  \
0           QKuHYh,vZGiwu2FWEJoLDx04  33-429-790-6131
1  ANDENSOSmk,miq23Xfb5RWt6dvUcvt6Qa  29-520-692-3537
2             B3rqp0xbSEim4Mpy2RH J  33-320-228-2957
```

```
3                      y3OD9UywSTOk  17-779-299-1839
4                        J8fcXWsTqM  17-813-485-8637

                                                S_COMMENT
0                     uriously regular requests hag
1  efully express instructions. regular requests …
2  etect about the furiously final accounts. slyl…
3  ackages boost blithely. blithely regular depos…
4  etect blithely bold asymptotes. fluffily ironi…
Size of query:  (460, 8)
```

[11]:
```
q3 = pd.read_csv("Q3.csv")
print(q3.head())
print("Size of query: ", q3.shape)
```

```
   L_ORDERKEY      revenue O_ORDERDATE  O_SHIPPRIORITY
0     5660420  389343.1796  1995-03-08               0
1     2118374  384042.9549  1995-03-19               0
2     3837441  382568.0383  1995-02-14               0
3      564738  378219.2915  1995-03-06               0
4     5006400  377083.4322  1995-03-18               0
Size of query:  (11264, 4)
```

[12]:
```
q4 = pd.read_csv("Q4.csv")
print(q4.head())
print("Size of query: ", q4.shape)
```

```
   O_ORDERPRIORITY  order_count
0         1-URGENT        10569
1           2-HIGH        10385
2         3-MEDIUM        10377
3  4-NOT SPECIFIED        10440
4            5-LOW        10393
Size of query:  (5, 2)
```

[13]:
```
q5 = pd.read_csv("Q5.csv")
print(q5.head())
print("Size of query: ", q5.shape)
```

```
           N_NAME       revenue
0         ROMANIA  5.483336e+07
1          RUSSIA  5.381822e+07
2         GERMANY  5.183593e+07
3          FRANCE  5.180258e+07
4  UNITED KINGDOM  4.910800e+07
Size of query:  (5, 2)
```

[14]:
```
q6 = pd.read_csv("Q6.csv")
print(q6.head())
```

```
print("Size of query: ", q6.shape)
```

```
       revenue
0  1.031663e+08
Size of query:  (1, 1)
```

[15]:
```
q7 = pd.read_csv("Q7.csv")
print(q7.head())
print("Size of query: ", q7.shape)
```

```
  supp_nation cust_nation  l_year       revenue
0        IRAQ        PERU    1995  5.659699e+07
1        IRAQ        PERU    1996  5.798743e+07
2        PERU        IRAQ    1995  5.141655e+07
3        PERU        IRAQ    1996  5.251922e+07
Size of query:  (4, 4)
```

[16]:
```
q8 = pd.read_csv("Q8.csv")
print(q8.head())
print("Size of query: ", q8.shape)
```

```
   o_year  mkt_share
0    1995   0.035234
1    1996   0.043039
Size of query:  (2, 2)
```

[17]:
```
q9 = pd.read_csv("Q9.csv")
print(q9.head())
print("Size of query: ", q9.shape)
```

```
    nation  o_year    sum_profit
0  ALGERIA    1998  2.901212e+07
1  ALGERIA    1997  4.943342e+07
2  ALGERIA    1996  4.560312e+07
3  ALGERIA    1995  4.631674e+07
4  ALGERIA    1994  4.859656e+07
Size of query:  (175, 3)
```

[18]:
```
q10 = pd.read_csv("Q10.csv")
print(q10.head())
print("Size of query: ", q10.shape)
```

```
   C_CUSTKEY             C_NAME      revenue  C_ACCTBAL         N_NAME  \
0      29518  Customer#000029518  695992.4969    9244.62        GERMANY
1      39412  Customer#000039412  691788.4046    8888.43           PERU
2      54826  Customer#000054826  668052.7708    1018.83          CHINA
3      59413  Customer#000059413  651289.9559    9948.30         CANADA
4     127486  Customer#000127486  648241.9962    6630.53  UNITED STATES

                          C_ADDRESS          C_PHONE  \
```

```
0           9l0hP7jIItXcPieiSIaKsQfiCn  17-567-804-8028
1                FYhhcuRk4eRpF,hylCoAP  27-684-509-6399
2  0qQN5BZXBDTUmndXQP,LO00e9ptdAWT3vMD  28-849-358-2535
3     YfRCT3dnnD3rwefu9r F1Rgl3a4nV4 x  13-419-750-2952
4                      4Cj9IM RXBb9JvRV  34-842-719-7704

                                        C_COMMENT
0                    wake. blithely ironic packages
1               l packages. special pinto beans are c
2  riously above the final packages. pinto beans …
3  nic, bold packages among the furiously regular…
4  ly ironic foxes. pending, ironic pinto beans d…
Size of query:  (37302, 8)
```

[20]:
```
q11 = pd.read_csv("Q11.csv")
print(q11.head())
print("Size of query: ", q11.shape)
```

```
   PS_PARTKEY        value
0       26964  19453391.03
1      181683  17650015.20
2      159774  17150761.56
3      181556  16128801.21
4      164951  15811680.50
Size of query:  (869, 2)
```

[22]:
```
q12 = pd.read_csv("Q12.csv")
print(q12.head())
print("Size of query: ", q12.shape)
```

```
  L_SHIPMODE  high_line_count  low_line_count
0        AIR             6293            9123
1       RAIL             6313            9320
Size of query:  (2, 3)
```

[24]:
```
q13 = pd.read_csv("Q13.csv")
print(q13.head())
print("Size of query: ", q13.shape)
```

```
   c_count  custdist
0        0     50004
1       10      6595
2        9      6595
3       11      6032
4        8      5923
Size of query:  (42, 2)
```

[26]:
```
q14 = pd.read_csv("Q14.csv")
print(q14.head())
```

```
print("Size of query: ", q14.shape)
```

```
   promo_revenue
0      16.417037
Size of query:  (1, 1)
```

[28]:
```
q15 = pd.read_csv("Q15.csv")
print(q15.head())
print("Size of query: ", q15.shape)
```

```
   S_SUPPKEY              S_NAME                   S_ADDRESS          S_PHONE  \
0       2932  Supplier#000002932  gAIrgSCdvtJltNKuZKRGYeYLRf  14-722-108-2914

   total_revenue
0   1.926296e+06
Size of query:  (1, 5)
```

[30]:
```
q16 = pd.read_csv("Q16.csv")
print(q16.head())
print("Size of query: ", q16.shape)
```

```
    P_BRAND                 P_TYPE  P_SIZE  supplier_cnt
0  Brand#41      MEDIUM BRUSHED TIN       3            28
1  Brand#11   STANDARD POLISHED TIN      39            24
2  Brand#13   MEDIUM BRUSHED NICKEL       1            24
3  Brand#21  MEDIUM ANODIZED COPPER       3            24
4  Brand#22    SMALL BRUSHED NICKEL       3            24
Size of query:  (18263, 4)
```

[32]:
```
q17 = pd.read_csv("Q17.csv")
print(q17.head())
print("Size of query: ", q17.shape)
```

```
     avg_yearly
0  344045.272857
Size of query:  (1, 1)
```

[33]:
```
q18 = pd.read_csv("Q18.csv")
print(q18.head())
print("Size of query: ", q18.shape)
```

```
               C_NAME  C_CUSTKEY  O_ORDERKEY O_ORDERDATE  O_TOTALPRICE  \
0  Customer#000128120     128120     4722021  1994-04-07     544089.09
1  Customer#000144617     144617     3043270  1997-02-12     530604.44
2  Customer#000066790      66790     2199712  1996-09-30     515531.82
3  Customer#000015619      15619     3767271  1996-08-07     480083.96
4  Customer#000147197     147197     1263015  1997-02-02     467149.67

   sum(l_quantity)
0              323
```

```
1              317
2              327
3              318
4              320
Size of query:  (9, 6)
```

[34]:
```
q19 = pd.read_csv("Q19.csv")
print(q19.head())
print("Size of query: ", q19.shape)
```

```
       revenue
0  3.039604e+06
Size of query:  (1, 1)
```

[35]:
```
q20 = pd.read_csv("Q20.csv")
print(q20.head())
print("Size of query: ", q20.shape)
```

```
               S_NAME                                  S_ADDRESS
0  Supplier#000000035         QymmGXxjVVQ5OuABCXVVsu,4eF gU0Qc6
1  Supplier#000000068  Ue6N50wH2CwE4PPgTGLmat,ibGYYlDoOb3xQwtgb
2  Supplier#000000080       cJ2MHSEJ13rIL2Wj3D5i6hRo30,ZiNUXhqn
3  Supplier#000000100               rIlN li8zvW22l2slbcx ECP4fL
4  Supplier#000000274                    usxbl9KSW41DTE6FAglxHU
Size of query:  (183, 2)
```

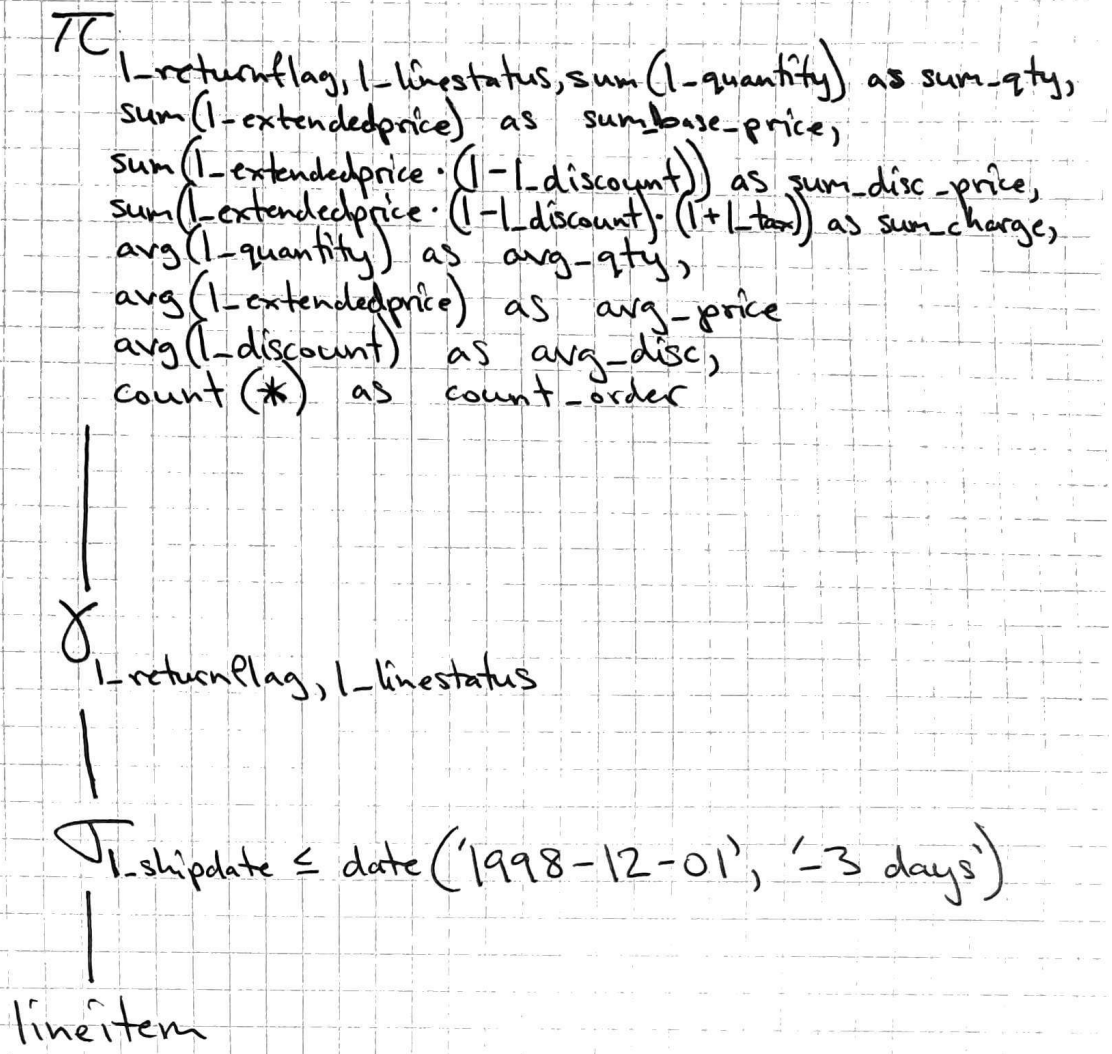
[36]:
```
q21 = pd.read_csv("Q21.csv")
print(q21.head())
print("Size of query: ", q21.shape)
```

```
               S_NAME  numwait
0  Supplier#000002340       25
1  Supplier#000004318       21
2  Supplier#000005601       19
3  Supplier#000009177       18
4  Supplier#000004804       17
Size of query:  (415, 2)
```

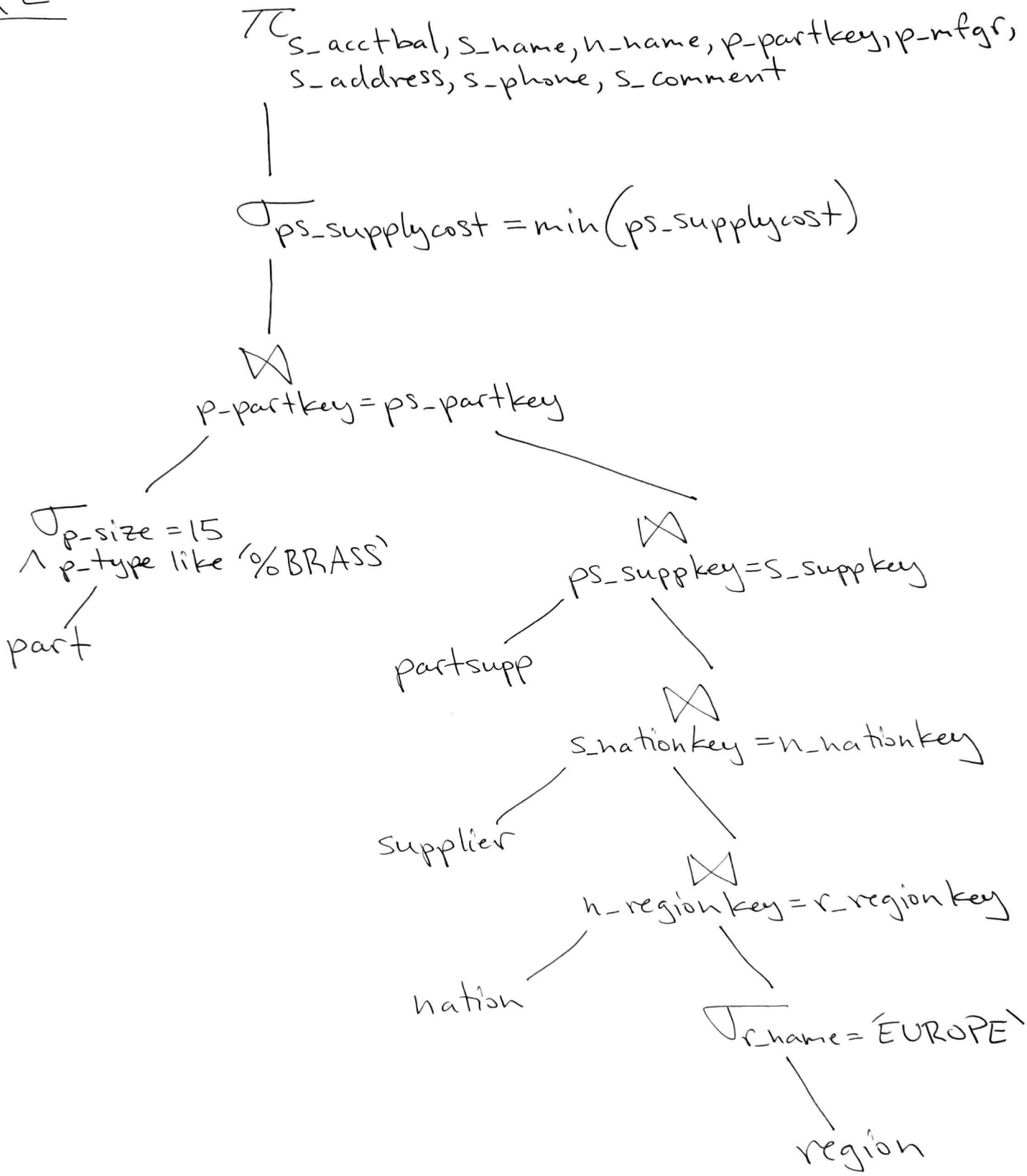
[37]:
```
q22 = pd.read_csv("Q22.csv")
print(q22.head())
print("Size of query: ", q22.shape)
```

```
   cntrycode  numcust  totacctbal
0         21      955  7235832.66
1         24      920  6866012.75
2         28      907  6700667.29
3         29      948  7158866.63
Size of query:  (4, 3)
```

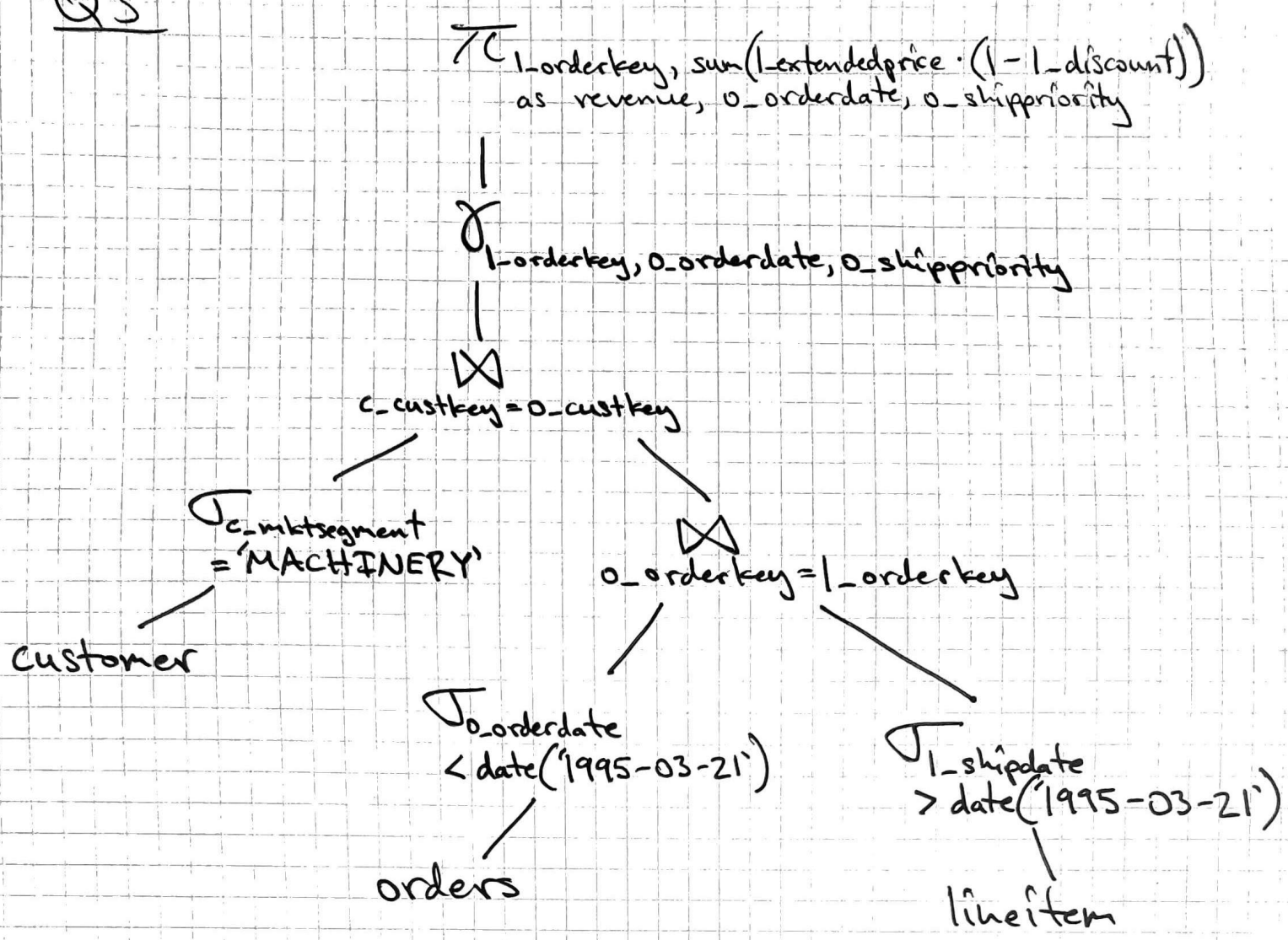
Q1


π
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
γ
l_returnflag, l_linestatus
σ
l_shipdate ≤ date('1998-12-01', '-3 days')
lineitem

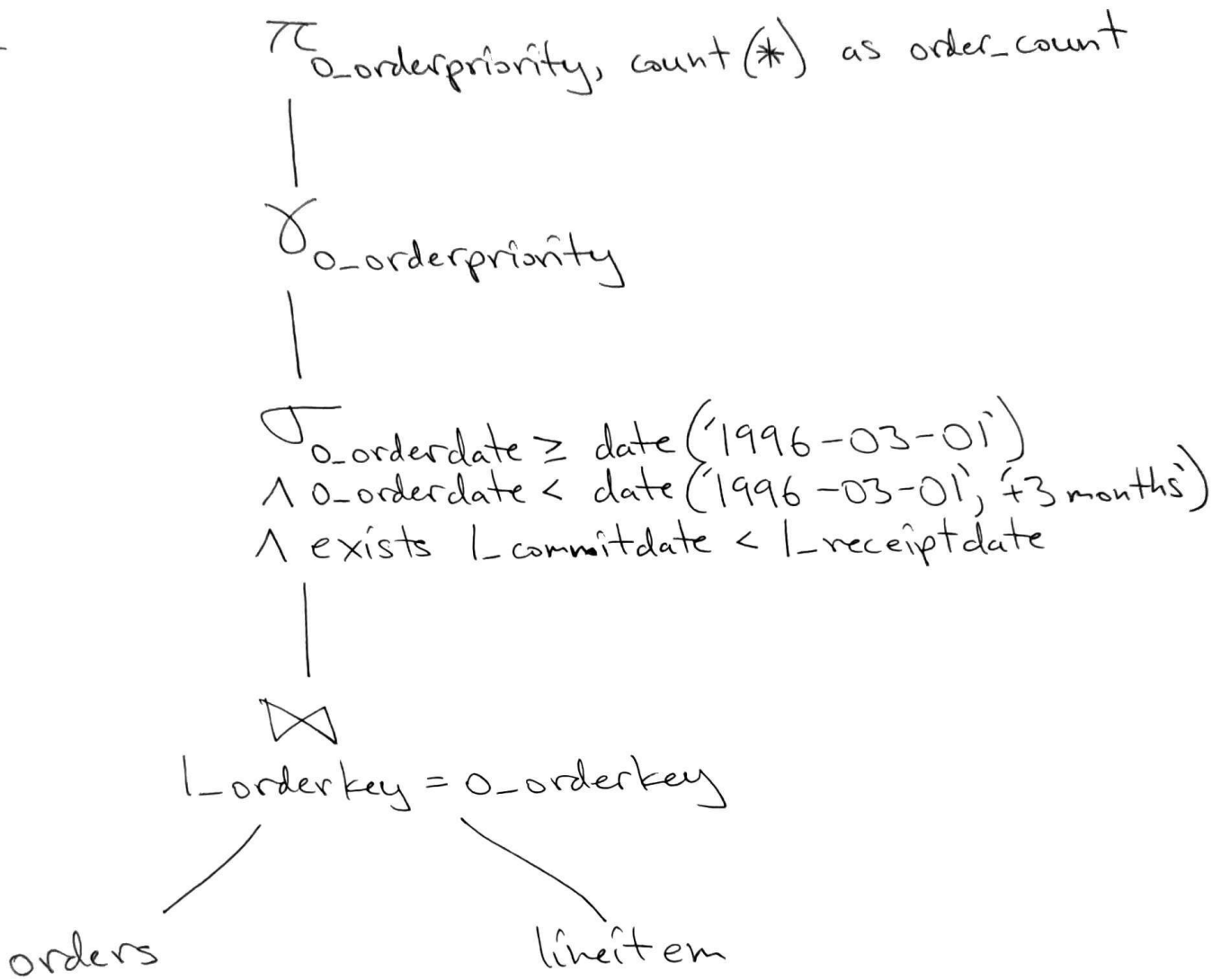
Q2


π s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
σ ps_supplycost = min(ps_supplycost)
⋈ p_partkey = ps_partkey
σ p_size = 15 ∧ p_type like '%BRASS'
part
⋈ ps_suppkey = s_suppkey
partsupp
⋈ s_nationkey = n_nationkey
supplier
⋈ n_regionkey = r_regionkey
nation
σ r_name = 'EUROPE'
region

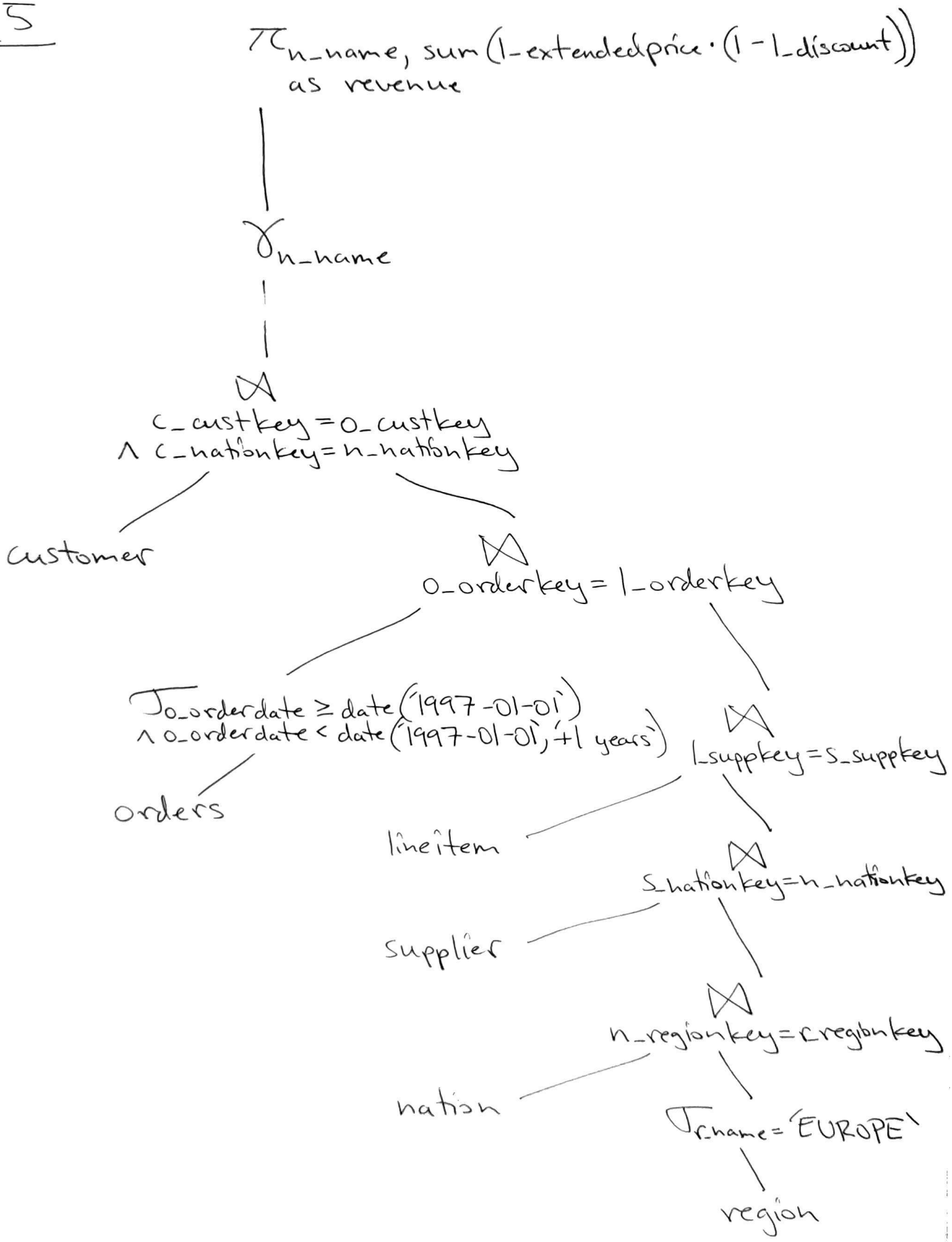
Q3


τ l_orderkey, sum(l_extendedprice · (1 − l_discount)) as revenue, o_orderdate, o_shippriority
γ l_orderkey, o_orderdate, o_shippriority
⋈ c_custkey = o_custkey
σ c_mktsegment = 'MACHINERY'
customer
⋈ o_orderkey = l_orderkey
σ o_orderdate < date('1995-03-21')
orders
σ l_shipdate > date('1995-03-21')
lineitem

Q4


π o_orderpriority, count(*) as order_count
γ o_orderpriority
σ o_orderdate ≥ date('1996-03-01')
∧ o_orderdate < date('1996-03-01', +3 months)
∧ exists l_commitdate < l_receiptdate
⋈
l_orderkey = o_orderkey
orders
lineitem

Q5


$\pi_{n\_name,\ sum(l\_extendedprice \cdot (1 - l\_discount))\ as\ revenue}$
γ_{n_name}
$\bowtie$
c_custkey = o_custkey
∧ c_nationkey = n_nationkey
customer
$\bowtie$
o_orderkey = l_orderkey
$\sigma_{o\_orderdate \geq date('1997-01-01') \wedge o\_orderdate < date('1997-01-01', '+1 years')}$
orders
$\bowtie$
l_suppkey = s_suppkey
lineitem
$\bowtie$
s_nationkey = n_nationkey
supplier
$\bowtie$
n_regionkey = r_regionkey
nation
$\sigma_{r\_name = 'EUROPE'}$
region

Q6

$\tau_{\text{sum}(l\_extendedprice \cdot l\_discount) \text{ as revenue}}$

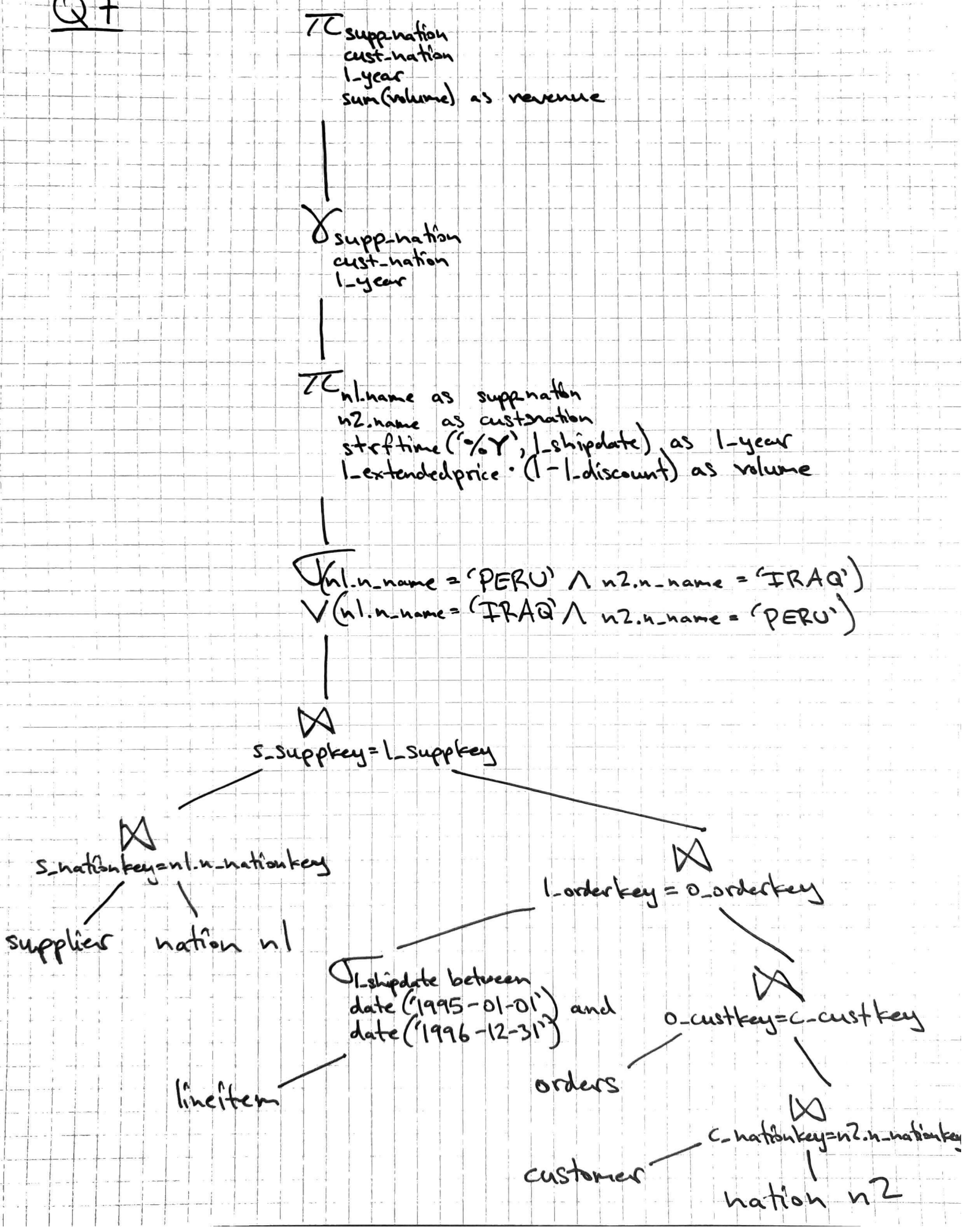
|

$\sigma_{l\_shipdate \geq date('1997-01-01')}$
$\wedge l\_shipdate < date('1997-01-01', '+1 years')$
$\wedge l\_discount$ between $0.07-0.01$ and $0.07+0.01$
$\wedge l\_quantity < 24$

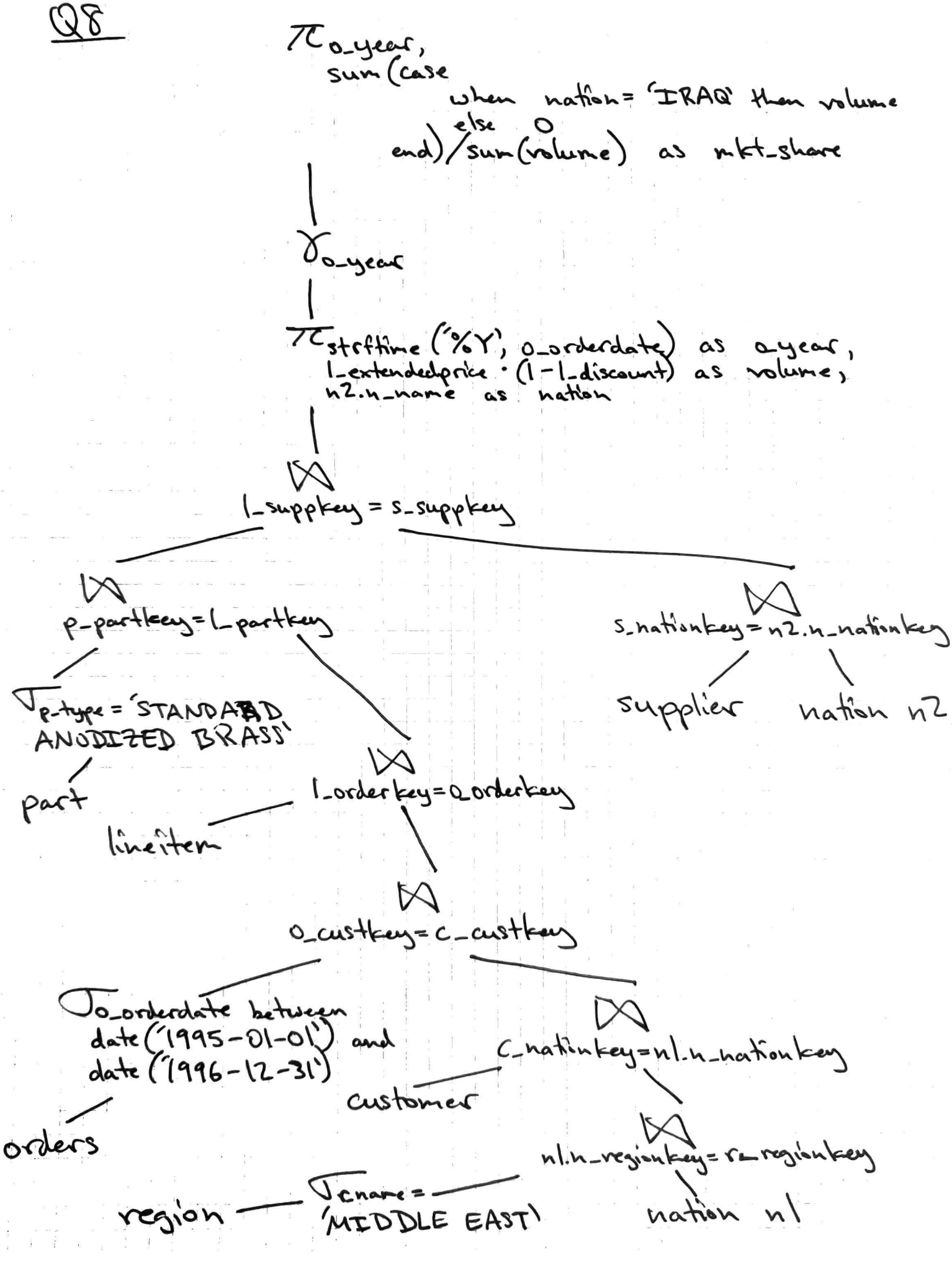
|

lineitem

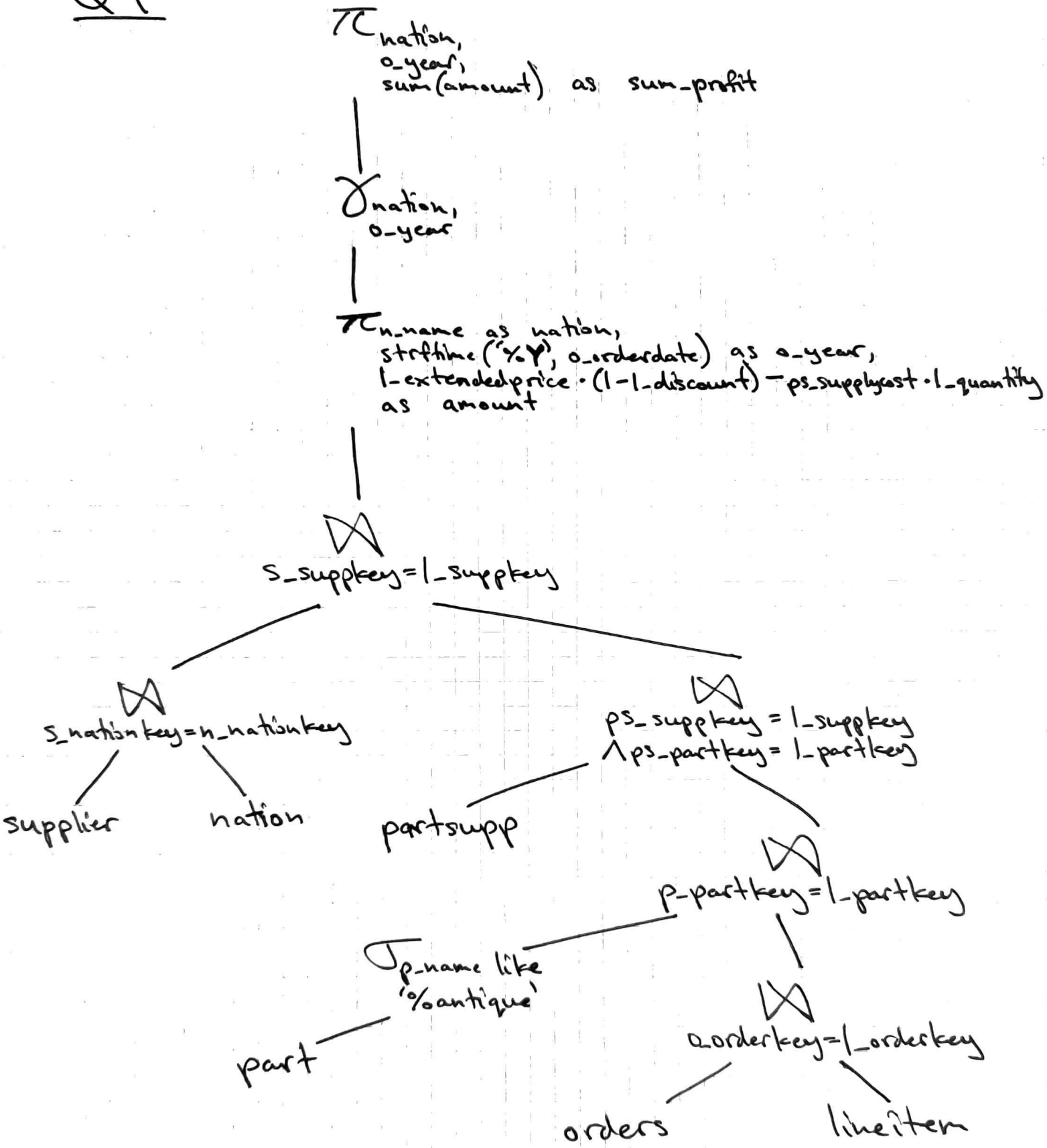
Q7


τ supp_nation
cust_nation
l_year
sum(volume) as revenue
γ supp_nation
cust_nation
l_year
τ n1.name as suppnation
n2.name as custnation
strftime('%Y', l_shipdate) as l_year
l_extendedprice · (1 - l_discount) as volume
σ (n1.n_name = 'PERU' ∧ n2.n_name = 'IRAQ')
∨ (n1.n_name = 'IRAQ' ∧ n2.n_name = 'PERU')
⋈ s_suppkey = l_suppkey
⋈ s_nationkey = n1.n_nationkey
suppliers
nation n1
⋈ l_orderkey = o_orderkey
σ l_shipdate between
date('1995-01-01') and
date('1996-12-31')
lineitem
⋈ o_custkey = c_custkey
orders
⋈ c_nationkey = n2.n_nationkey
customer
nation n2

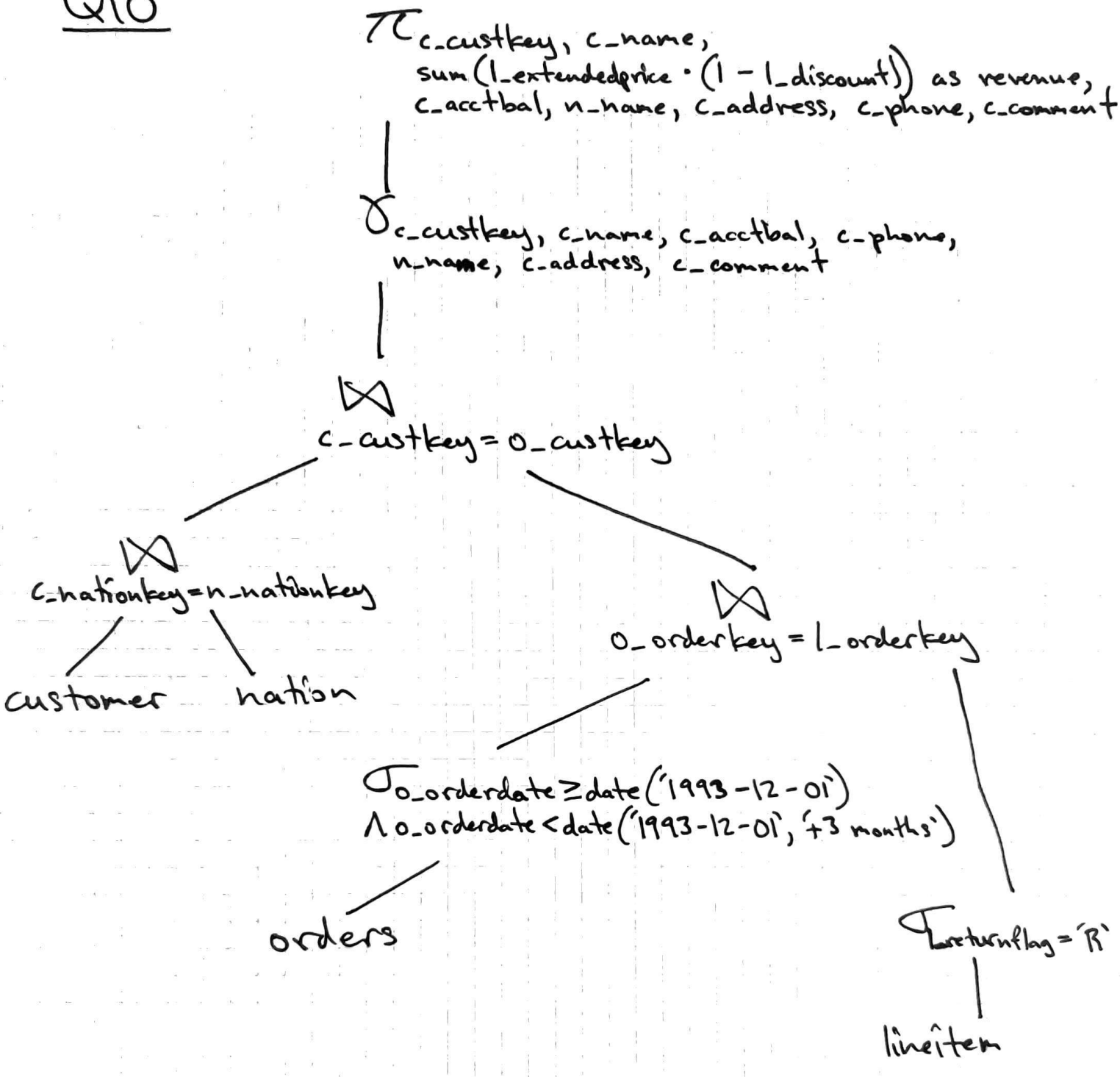
Q8


π o_year, sum(case when nation = 'IRAQ' then volume else 0 end) / sum(volume) as mkt_share
γ o_year
π strftime('%Y', o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume, n2.n_name as nation
⋈ l_suppkey = s_suppkey
⋈ p_partkey = l_partkey
⋈ s_nationkey = n2.n_nationkey
σ p_type = 'STANDARD ANODIZED BRASS'
part
lineitem
⋈ l_orderkey = o_orderkey
supplier
nation n2
⋈ o_custkey = c_custkey
σ o_orderdate between date('1995-01-01') and date('1996-12-31')
orders
⋈ c_nationkey = n1.n_nationkey
customer
⋈ n1.n_regionkey = r_regionkey
σ r_name = 'MIDDLE EAST'
region
nation n1

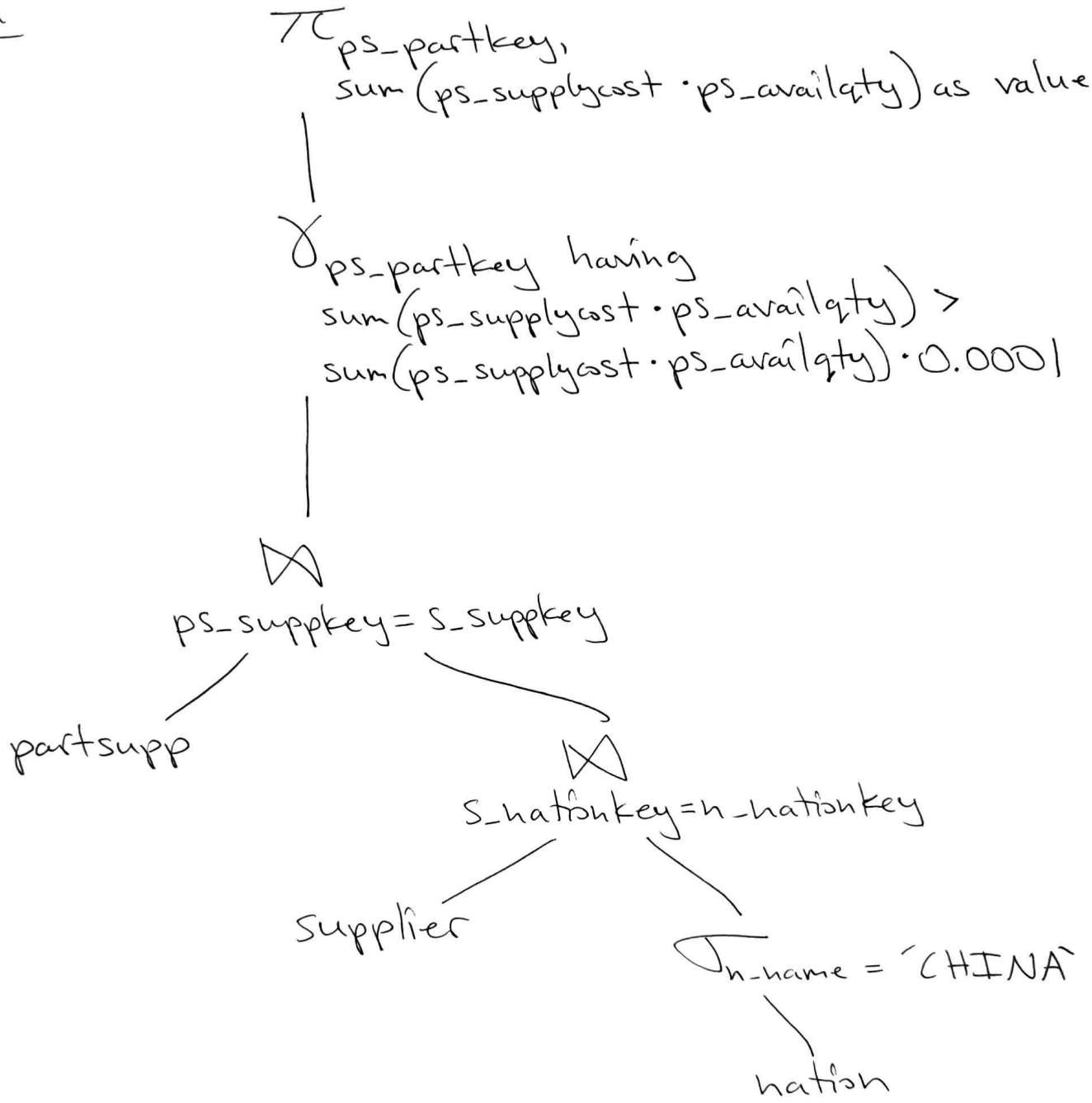
Q9


π nation, o_year, sum(amount) as sum_profit
γ nation, o_year
π n_name as nation, strftime('%Y', o_orderdate) as o_year, l_extendedprice · (1 - l_discount) - ps_supplycost · l_quantity as amount
⋈ s_suppkey = l_suppkey
⋈ s_nationkey = n_nationkey
supplier
nation
⋈ ps_suppkey = l_suppkey ∧ ps_partkey = l_partkey
partsupp
⋈ p_partkey = l_partkey
σ p_name like '%antique'
part
⋈ o_orderkey = l_orderkey
orders
lineitem

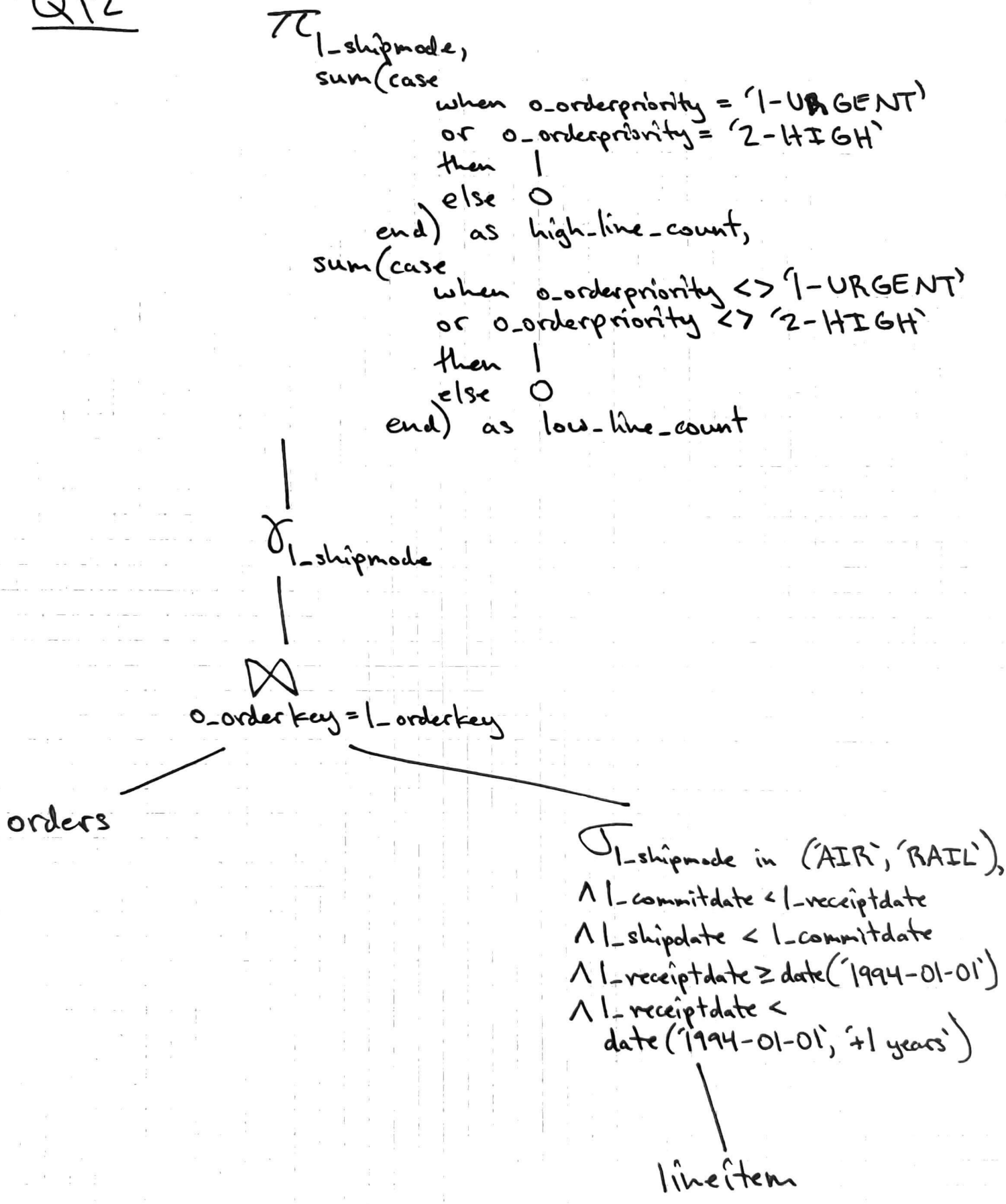
Q10


π c_custkey, c_name,
sum(l_extendedprice · (1 − l_discount)) as revenue,
c_acctbal, n_name, c_address, c_phone, c_comment
γ c_custkey, c_name, c_acctbal, c_phone,
n_name, c_address, c_comment
⋈
c_custkey = o_custkey
⋈
c_nationkey = n_nationkey
customer
nation
⋈
o_orderkey = l_orderkey
σ o_orderdate ≥ date('1993-12-01')
∧ o_orderdate < date('1993-12-01', '+3 months')
orders
σ l_returnflag = 'R'
lineitem

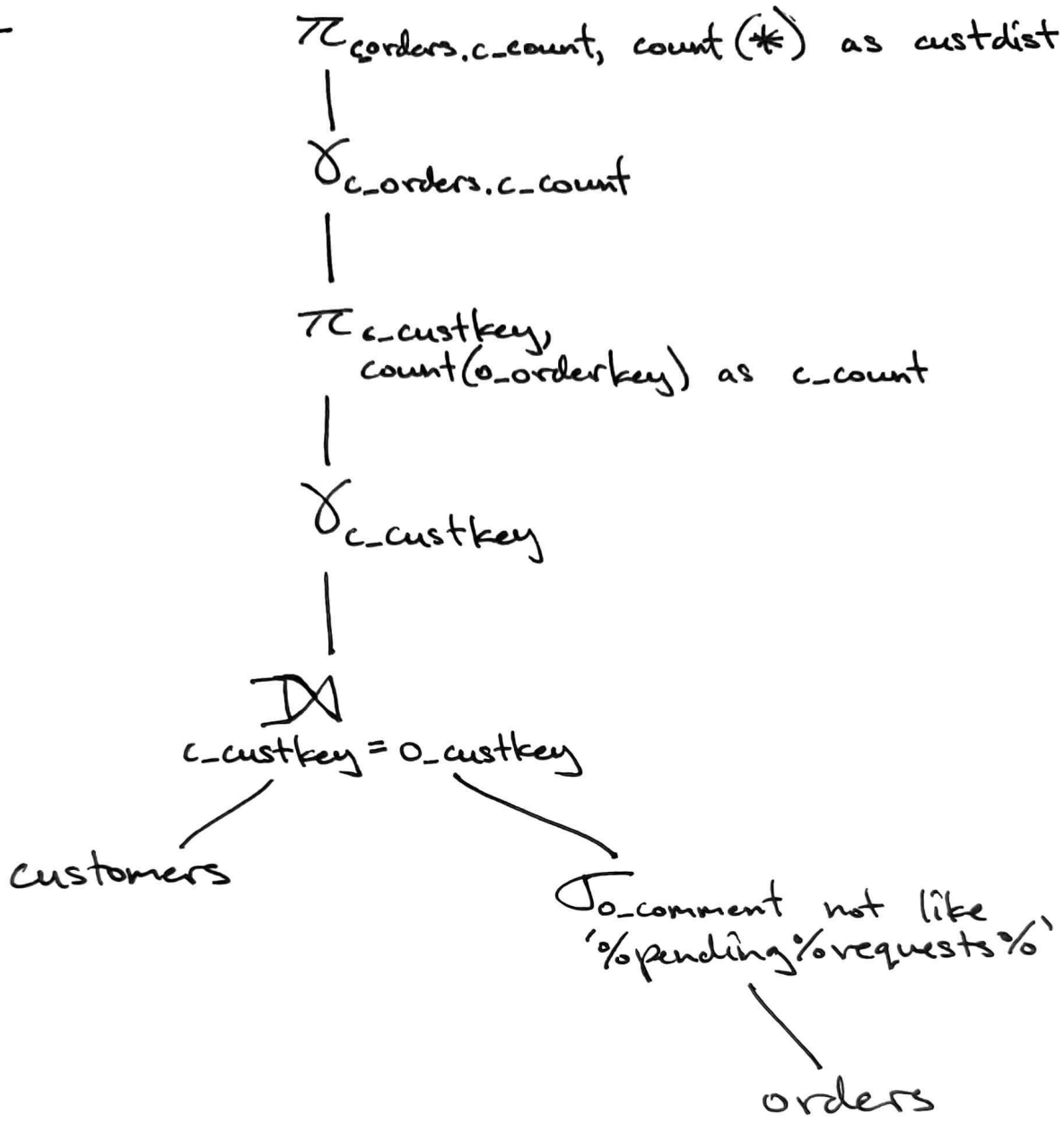
Q11


π ps_partkey, sum(ps_supplycost · ps_availqty) as value
γ ps_partkey having sum(ps_supplycost · ps_availqty) > sum(ps_supplycost · ps_availqty) · 0.0001
⋈ ps_suppkey = s_suppkey
partsupp
⋈ s_nationkey = n_nationkey
supplier
σ n_name = 'CHINA'
nation

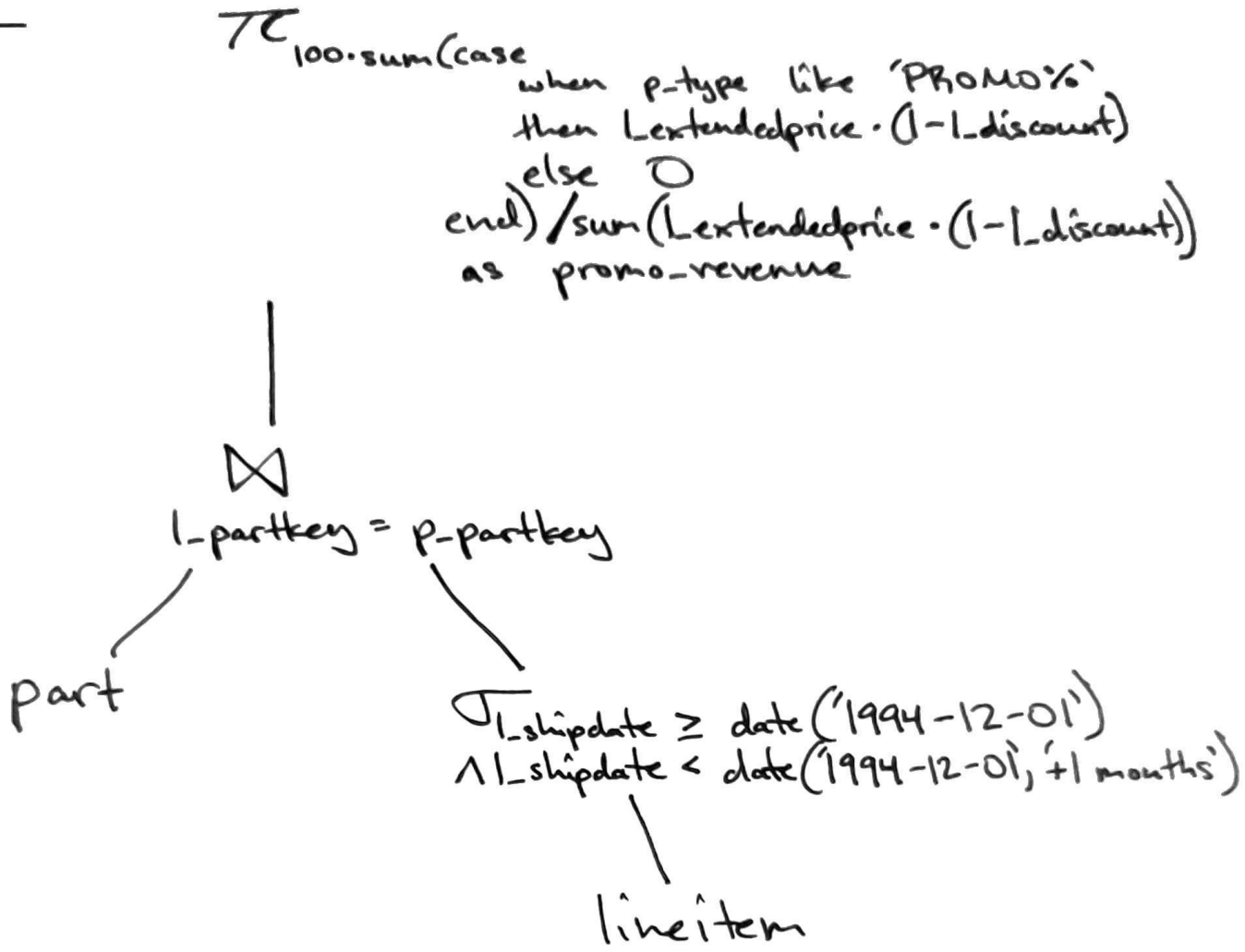
Q12


π l_shipmode,
sum(case
when o_orderpriority = '1-URGENT'
or o_orderpriority = '2-HIGH'
then 1
else 0
end) as high_line_count,
sum(case
when o_orderpriority <> '1-URGENT'
or o_orderpriority <> '2-HIGH'
then 1
else 0
end) as low_line_count
γ l_shipmode
⋈
o_orderkey = l_orderkey
orders
σ l_shipmode in ('AIR', 'RAIL'),
∧ l_commitdate < l_receiptdate
∧ l_shipdate < l_commitdate
∧ l_receiptdate ≥ date('1994-01-01')
∧ l_receiptdate <
date('1994-01-01', '+1 years')
lineitem

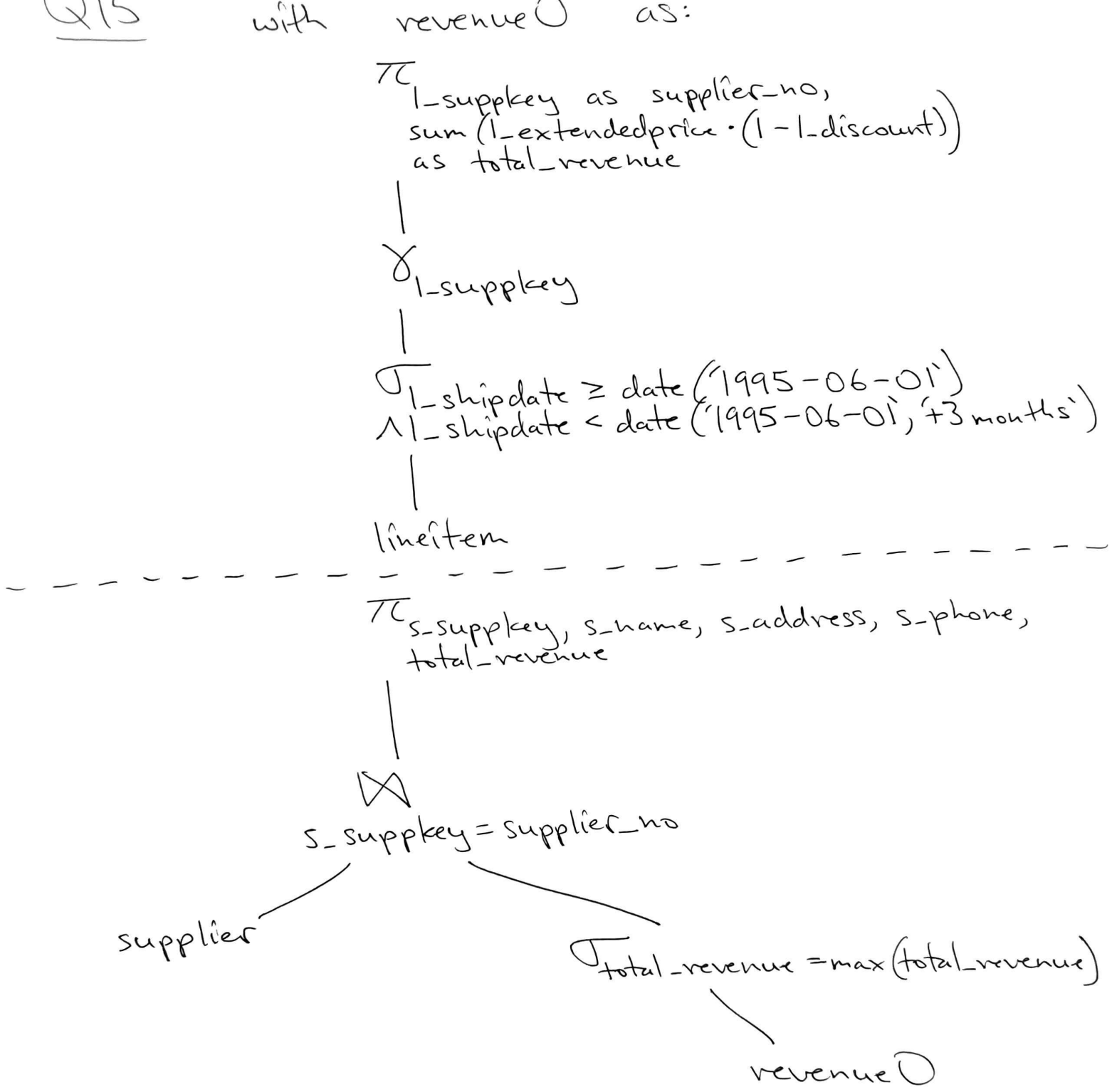
Q13


π c_orders.c_count, count(*) as custdist
γ c_orders.c_count
π c_custkey, count(o_orderkey) as c_count
γ c_custkey
⋈
c_custkey = o_custkey
customers
σ o_comment not like '%pending%requests%'
orders

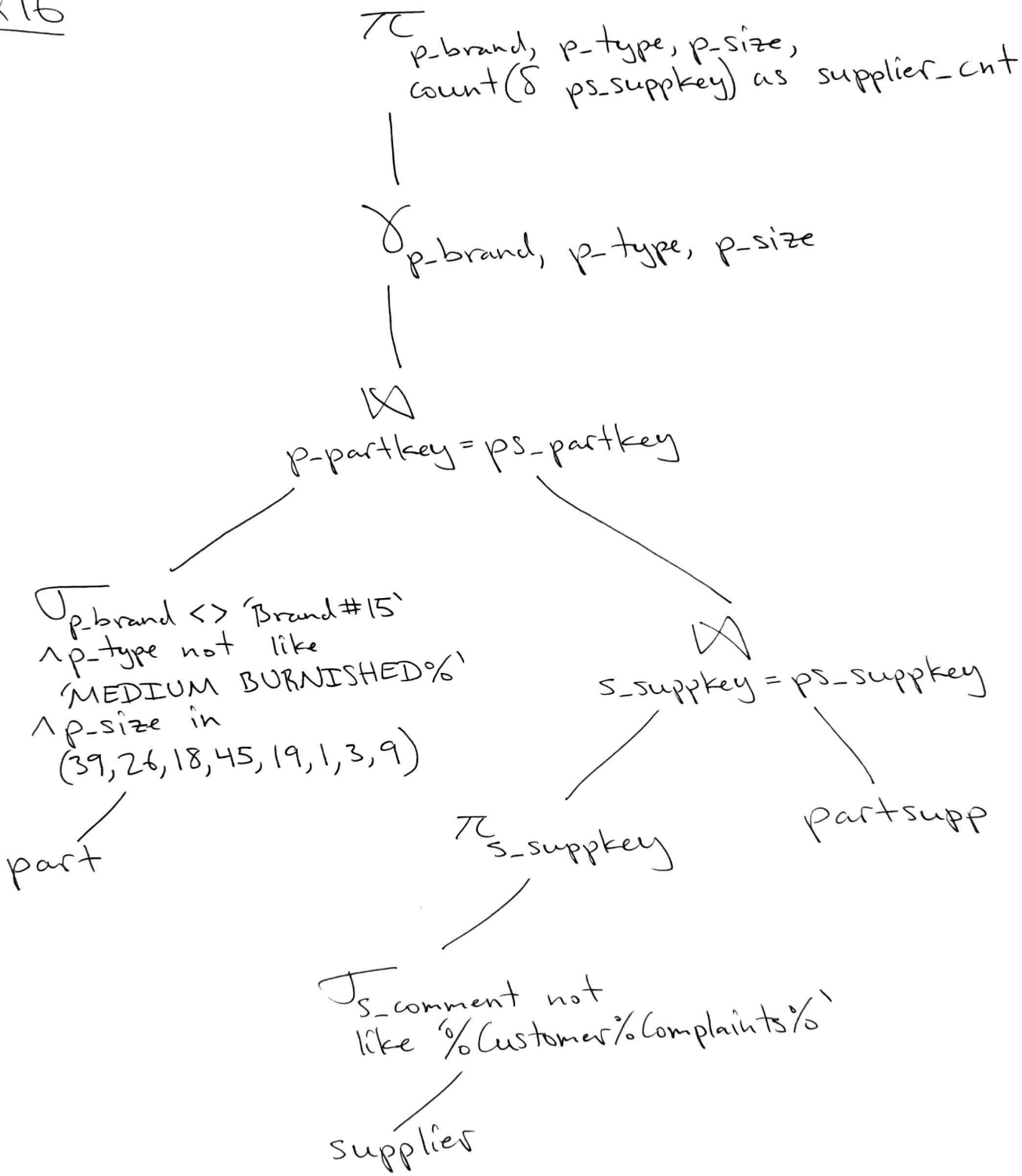
Q14


π 100.sum(case
when p_type like 'PROMO%'
then l_extendedprice · (1-l_discount)
else 0
end) / sum(l_extendedprice · (1-l_discount))
as promo_revenue
⋈
l_partkey = p_partkey
part
σ l_shipdate ≥ date('1994-12-01')
∧ l_shipdate < date('1994-12-01', '+1 months')
lineitem

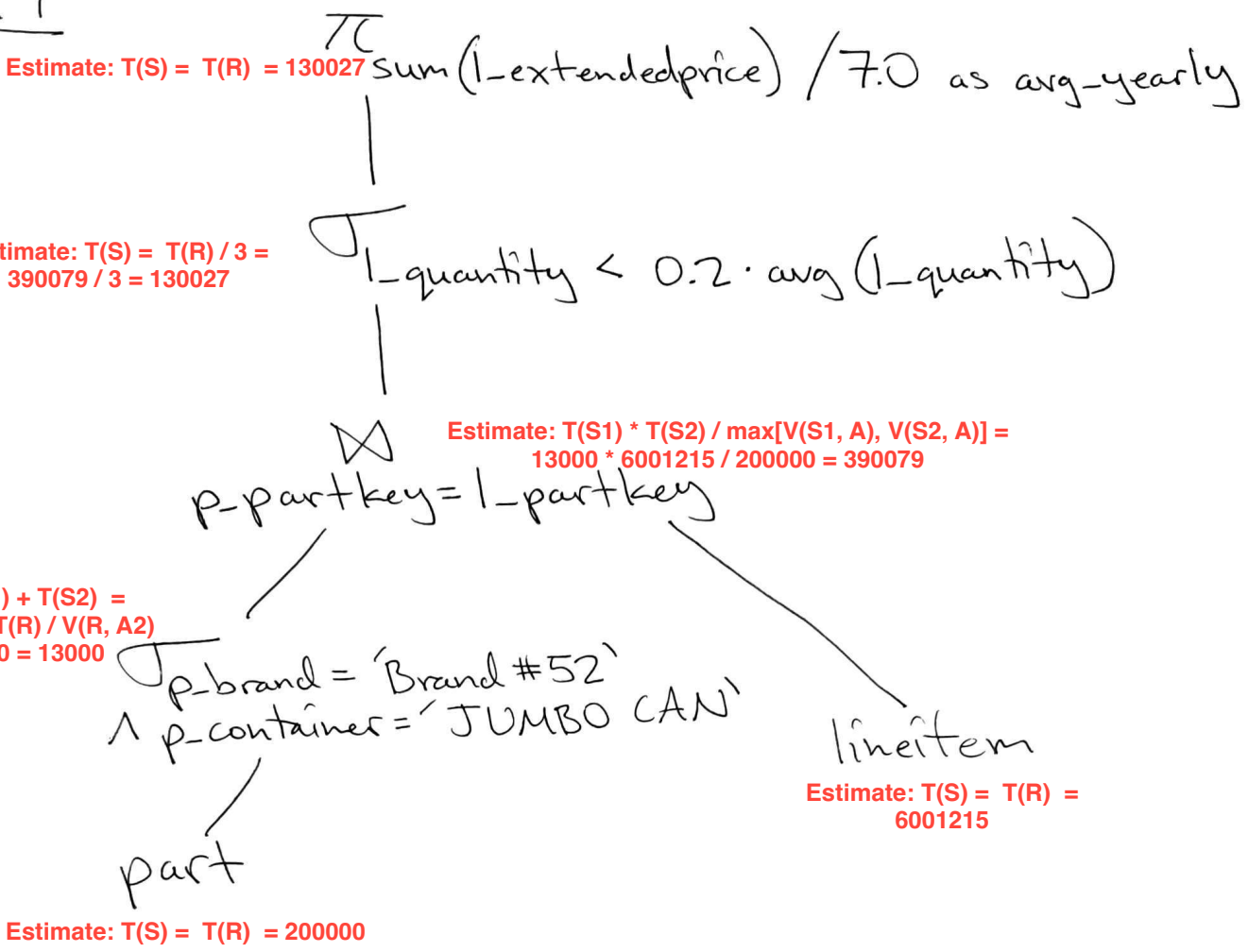
Q15


with revenue0 as:
π
l_suppkey as supplier_no,
sum(l_extendedprice · (1 - l_discount))
as total_revenue
γ
l_suppkey
σ
l_shipdate ≥ date('1995-06-01')
∧ l_shipdate < date('1995-06-01', '+3 months')
lineitem
π
s_suppkey, s_name, s_address, s_phone,
total_revenue
⋈
s_suppkey = supplier_no
supplier
σ
total_revenue = max(total_revenue)
revenue0

Q16


π p_brand, p_type, p_size, count(δ ps_suppkey) as supplier_cnt
γ p_brand, p_type, p_size
⋈ p_partkey = ps_partkey
σ p_brand <> 'Brand#15' ^ p_type not like 'MEDIUM BURNISHED%' ^ p_size in (39,26,18,45,19,1,3,9)
part
⋈ s_suppkey = ps_suppkey
π s_suppkey
partsupp
σ s_comment not like '%Customer%Complaints%'
supplier

Q17



Estimated Query Total: 13000 + 200000 + 6001215 + 390079 + 130027 + 130027 = 6,864,348

Q18


π c_name, c_custkey, o_orderkey, o_totalprice, sum(l_quantity), o_orderdate
γ c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
⋈ c_custkey = o_custkey
customer
⋈ o_orderkey = l_orderkey
orders
γ l_orderkey having sum(l_quantity) > 313
lineitem

Q19

$\pi_{sum(l\_extendedprice \cdot (1 - l\_discount))\ as\ revenue}$

|

$\bowtie$

(p_partkey = l_partkey
∧ p_brand = 'Brand #43'
∧ p_container in ('SM CASE', 'SM BOX',
'SM PACK', 'SM PKG')
∧ p_size between 1 and 5
∧ l_quantity ≥ 3
∧ l_quantity ≤ 3 + 10)

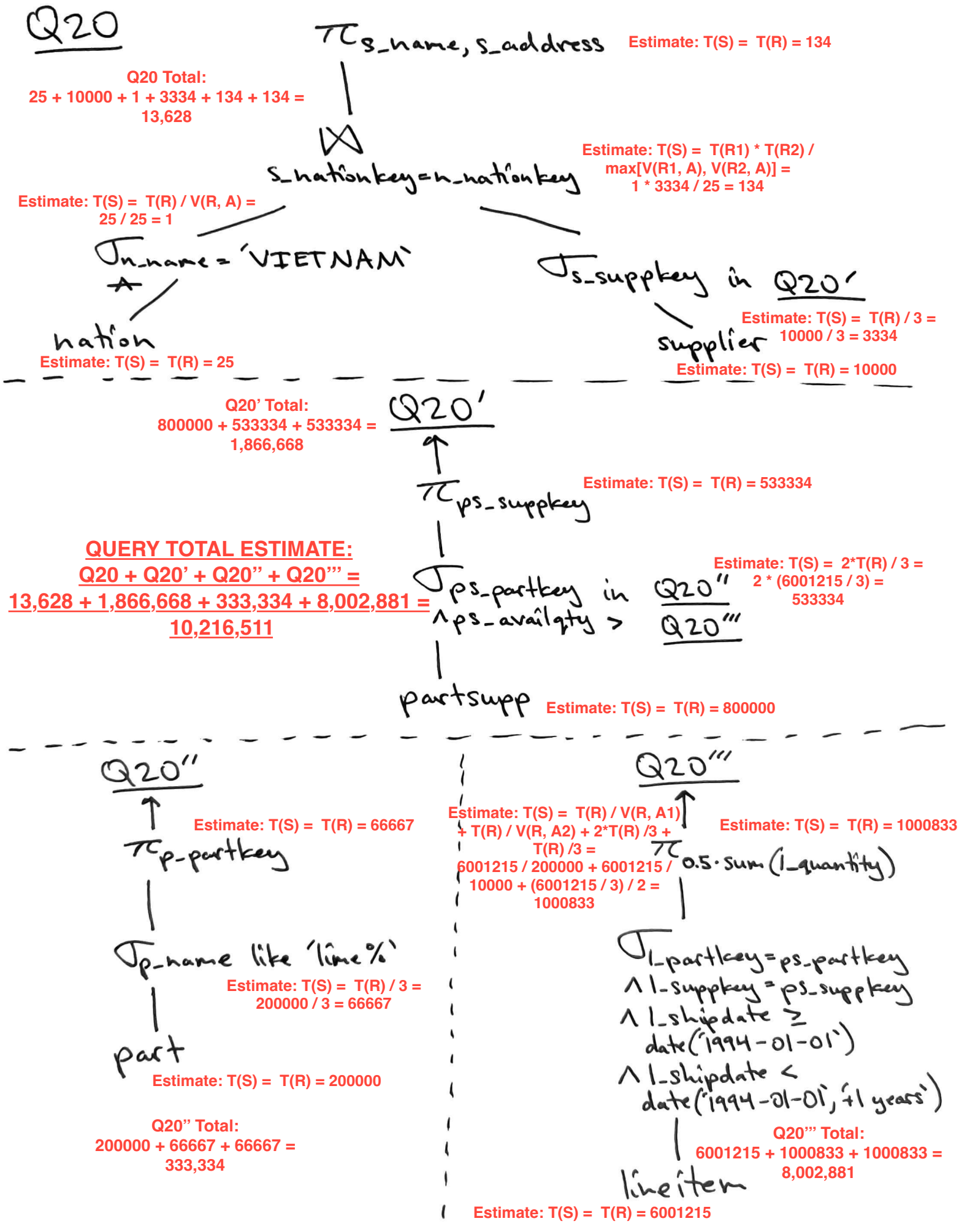
∨ (p_partkey = l_partkey
∧ p_brand = 'Brand #25'
∧ p_container in ('MED BAG', 'MED BOX',
'MED PKG', 'MED PACK')
∧ p_size between 1 and 10
∧ l_quantity ≥ 10
∧ l_quantity ≤ 10 + 10)

∨ (p_partkey = l_partkey
∧ p_brand = 'Brand #24'
∧ p_container in ('LG CASE', 'LG BOX',
'LG PACK', 'LG PKG')
∧ p_size between 1 and 15
∧ l_quantity ≥ 22
∧ l_quantity ≤ 22 + 10)

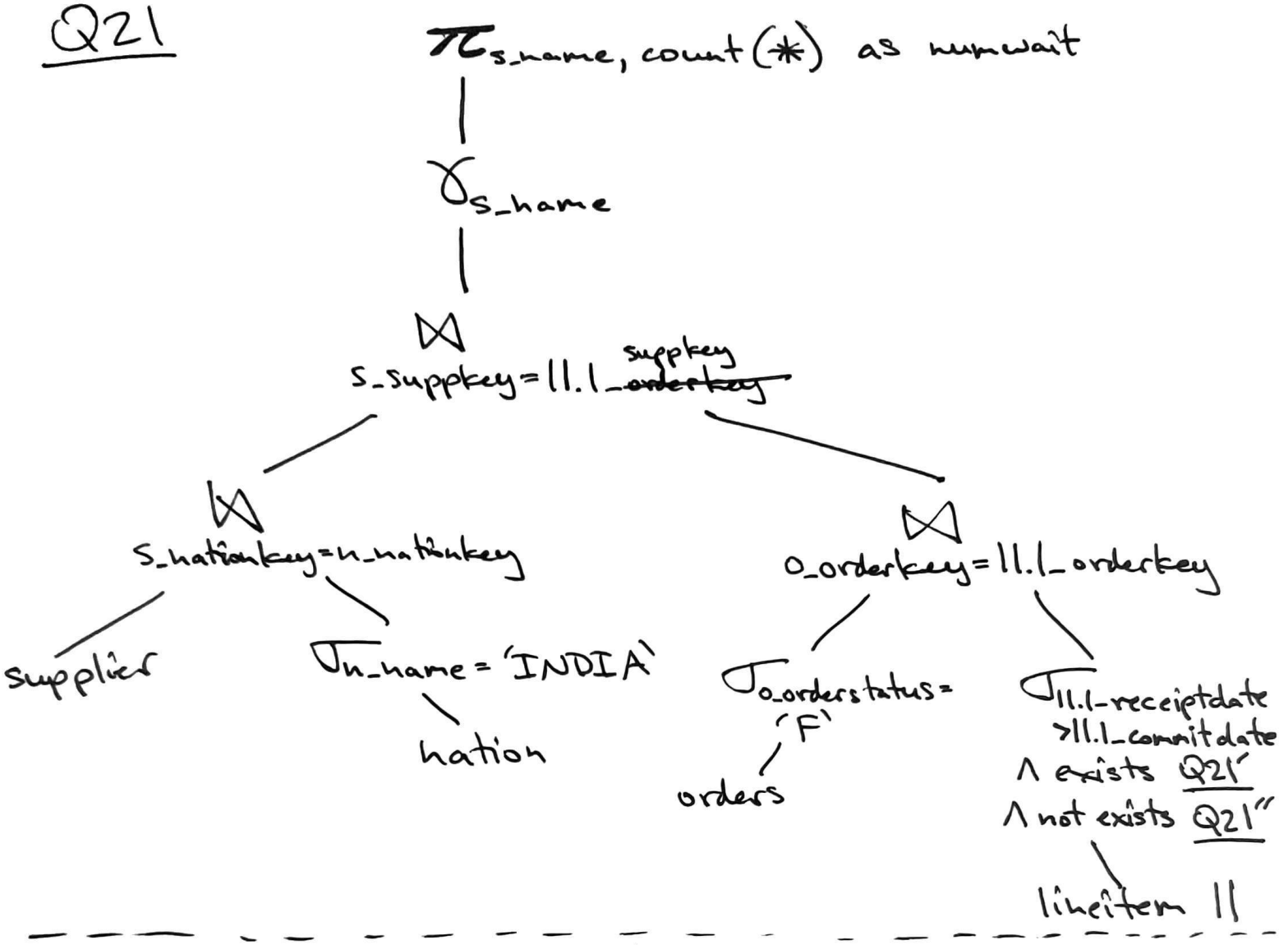
part

lineitem

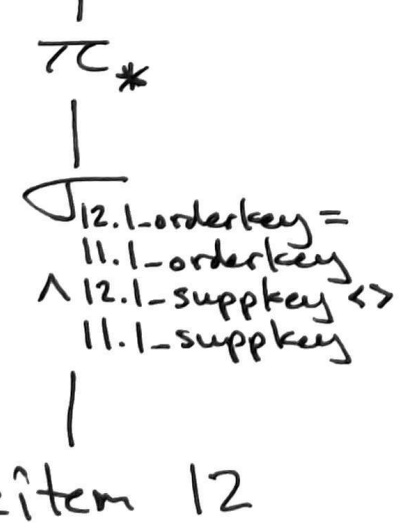
$\sigma_{l\_shipmode\ in\ ('AIR', 'AIR\ REG')}$
∧ l_shipinstruct = 'DELIVER IN PERSON'


Q20
$\pi_{s\_name, s\_address}$
Estimate: T(S) = T(R) = 134
Q20 Total:
25 + 10000 + 1 + 3334 + 134 + 134 =
13,628
$\bowtie$
s_nationkey=n_nationkey
Estimate: T(S) = T(R1) * T(R2) /
max[V(R1, A), V(R2, A)] =
1 * 3334 / 25 = 134
Estimate: T(S) = T(R) / V(R, A) =
25 / 25 = 1
$\sigma_{n\_name = 'VIETNAM'}$
(A)
nation
Estimate: T(S) = T(R) = 25
$\sigma_{s\_suppkey \text{ in } Q20'}$
Estimate: T(S) = T(R) / 3 =
10000 / 3 = 3334
supplier
Estimate: T(S) = T(R) = 10000
Q20' Total:
800000 + 533334 + 533334 =
1,866,668
Q20'
Estimate: T(S) = T(R) = 533334
$\pi_{ps\_suppkey}$
QUERY TOTAL ESTIMATE:
Q20 + Q20' + Q20'' + Q20''' =
13,628 + 1,866,668 + 333,334 + 8,002,881 =
10,216,511
$\sigma_{ps\_partkey \text{ in } Q20'' \wedge ps\_availqty > Q20'''}$
Estimate: T(S) = 2*T(R) / 3 =
2 * (6001215 / 3) =
533334
partsupp
Estimate: T(S) = T(R) = 800000
Q20''
Estimate: T(S) = T(R) = 66667
$\pi_{p\_partkey}$
$\sigma_{p\_name \text{ like } 'lime\%'}$
Estimate: T(S) = T(R) / 3 =
200000 / 3 = 66667
part
Estimate: T(S) = T(R) = 200000
Q20'' Total:
200000 + 66667 + 66667 =
333,334
Q20'''
Estimate: T(S) = T(R) / V(R, A1)
+ T(R) / V(R, A2) + 2*T(R) /3 +
T(R) /3 =
6001215 / 200000 + 6001215 /
10000 + (6001215 / 3) / 2 =
1000833
Estimate: T(S) = T(R) = 1000833
$\gamma_{0.5 \cdot sum(l\_quantity)}$
$\sigma_{l\_partkey = ps\_partkey \wedge l\_suppkey = ps\_suppkey \wedge l\_shipdate \geq date('1994-01-01') \wedge l\_shipdate < date('1994-01-01', '+1 years')}$
Q20''' Total:
6001215 + 1000833 + 1000833 =
8,002,881
lineitem
Estimate: T(S) = T(R) = 6001215

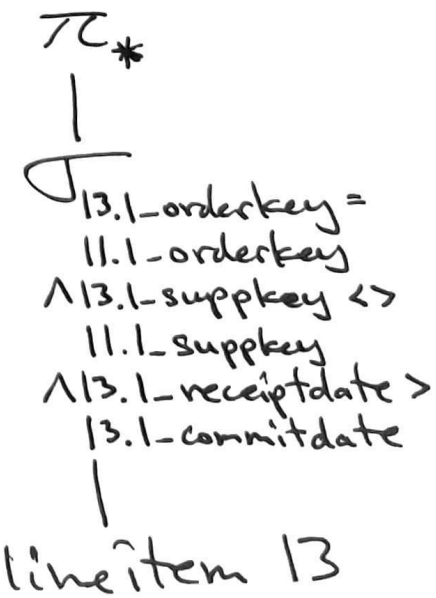
Q21

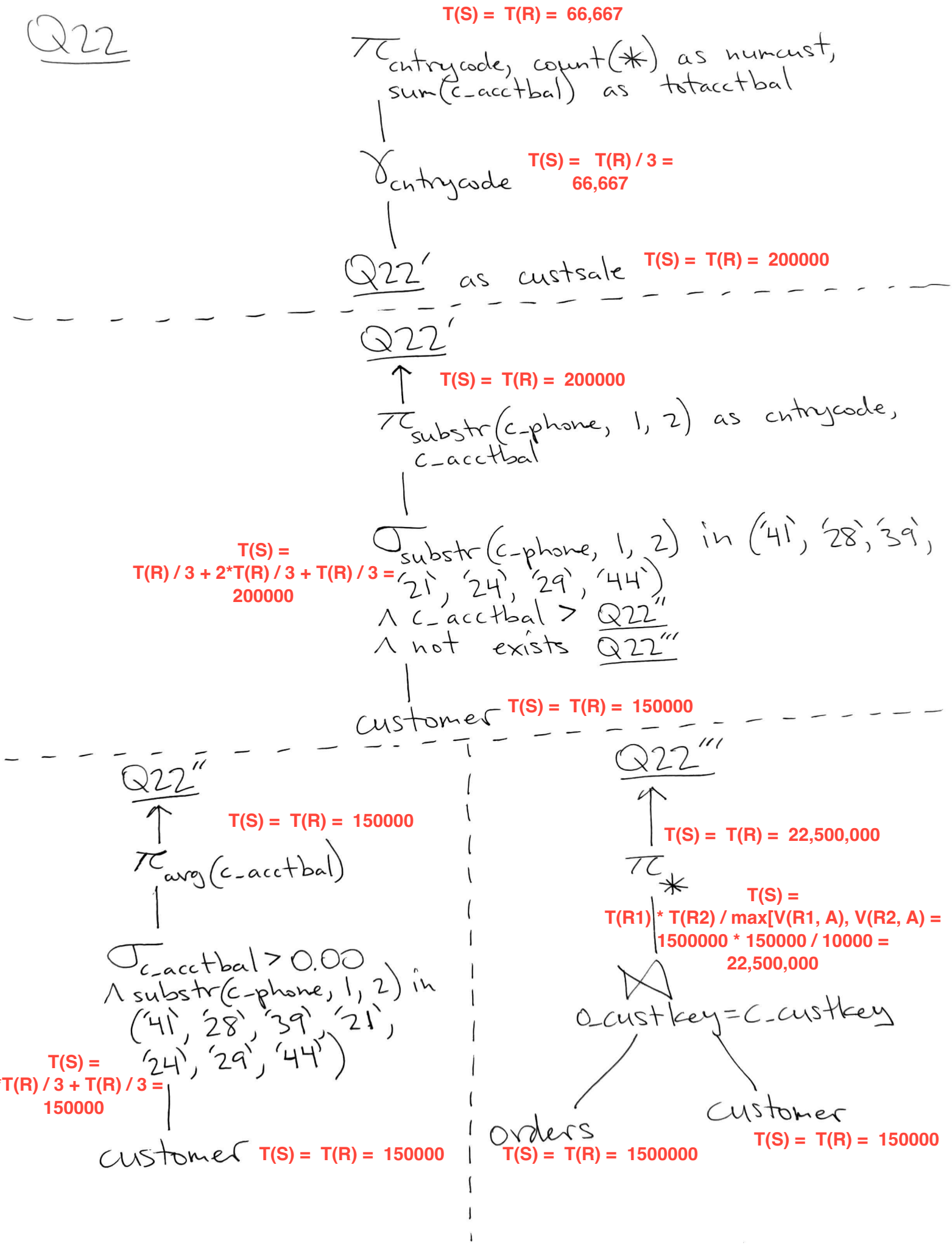

$\pi_{s\_name,\ count(*)\ as\ numwait}$
γ_{s_name}
$\bowtie$
s_suppkey=l1.l_suppkey
$\bowtie$
s_nationkey=n_nationkey
supplier
$\sigma_{n\_name='INDIA'}$
nation
$\bowtie$
o_orderkey=l1.l_orderkey
$\sigma_{o\_orderstatus='F'}$
orders
$\sigma_{l1.l\_receiptdate > l1.l\_commitdate \wedge exists\ Q21' \wedge not\ exists\ Q21''}$
lineitem l1

Q21'


π_*
$\sigma_{l2.l\_orderkey = l1.l\_orderkey \wedge l2.l\_suppkey <> l1.l\_suppkey}$
lineitem l2

Q21''


π_*
$\sigma_{l3.l\_orderkey = l1.l\_orderkey \wedge l3.l\_suppkey <> l1.l\_suppkey \wedge l3.l\_receiptdate > l3.l\_commitdate}$
lineitem l3


Q22
T(S) = T(R) = 66,667
πcntrycode, count(*) as numcust, sum(c_acctbal) as totacctbal
γcntrycode
T(S) = T(R) / 3 = 66,667
Q22' as custsale
T(S) = T(R) = 200000
Q22'
T(S) = T(R) = 200000
πsubstr(c_phone, 1, 2) as cntrycode, c_acctbal
σsubstr(c_phone, 1, 2) in ('41', '28', '39', '21', '24', '29', '44') ∧ c_acctbal > Q22'' ∧ not exists Q22'''
T(S) = T(R) / 3 + 2*T(R) / 3 + T(R) / 3 = 200000
customer
T(S) = T(R) = 150000
Q22''
T(S) = T(R) = 150000
πavg(c_acctbal)
σc_acctbal > 0.00 ∧ substr(c_phone, 1, 2) in ('41', '28', '39', '21', '24', '29', '44')
T(S) = 2*T(R) / 3 + T(R) / 3 = 150000
customer
T(S) = T(R) = 150000
Q22'''
T(S) = T(R) = 22,500,000
π*
T(S) = T(R1) * T(R2) / max[V(R1, A), V(R2, A) = 1500000 * 150000 / 10000 = 22,500,000
o_custkey=c_custkey
orders
T(S) = T(R) = 1500000
customer
T(S) = T(R) = 150000

Cardinality Estimations

April 16, 2024

[1]:
```
import pandas as pd
import sqlite3
```

[2]:
```
db_path = '/Users/tylernardone/Desktop/tpch.db'
```

[3]:
```
conn = sqlite3.connect(db_path)
```

[4]:
```
cursor = conn.cursor()
```

The dictionary below will serve as the basis for creating a dataframe that will store the cardinality estimates of each relation.

[5]:
```
relations = {
    'part': ['p_partkey',
             'p_name',
             'p_mfgr',
             'p_brand',
             'p_type',
             'p_size',
             'p_container',
             'p_retailprice',
             'p_comment'],

    'supplier': ['s_suppkey',
                 's_name',
                 's_address',
                 's_nationkey',
                 's_phone',
                 's_acctbal',
                 's_comment'],

    'partsupp': ['ps_partkey',
                 'ps_suppkey',
                 'ps_availqty',
                 'ps_supplycost',
                 'ps_comment'],

    'customer': ['c_custkey',
```

```
                'c_name',
                'c_address',
                'c_nationkey',
                'c_phone',
                'c_acctbal',
                'c_mktsegment',
                'c_comment'],

    'nation': ['n_nationkey',
               'n_name',
               'n_regionkey',
               'n_comment'],

    'lineitem': ['l_orderkey',
                 'l_partkey',
                 'l_suppkey',
                 'l_linenumber',
                 'l_quantity',
                 'l_extendedprice',
                 'l_discount',
                 'l_tax',
                 'l_returnflag',
                 'l_linestatus',
                 'l_shipdate',
                 'l_commitdate',
                 'l_receiptdate',
                 'l_shipinstruct',
                 'l_shipmode',
                 'l_comment'],

    'region': ['r_regionkey',
               'r_name',
               'r_comment'],

    'orders': ['o_orderkey',
               'o_custkey',
               'o_orderstatus',
               'o_totalprice',
               'o_orderdate',
               'o_orderpriority',
               'o_clerk',
               'o_shippriority',
               'o_comment'],
}
```

[6]:
```
n_occurrences = [len(list(relations.values())[i]) for i in range(len(relations.
↪values()))]
```

[7]:
```
all_relations = [
    [j] * n_occurrences[i] for i, j in enumerate(relations.keys())
]
```

[8]:
```
flattened_relations = [att for lst in all_relations for att in lst]
```

[9]:
```
all_attributes = [relations[i] for i in relations.keys()]
```

[10]:
```
flattened_attributes = [att for lst in all_attributes for att in lst]
```

[11]:
```
relation_df = pd.DataFrame(
    [flattened_relations, flattened_attributes]
).T.rename(
    {0: 'Relation',
     1: 'Attribute'
    }, axis=1
).set_index(['Relation', 'Attribute'])
```

[12]:
```
relation_df
```

[12]:
```
Empty DataFrame
Columns: []
Index: [(part, p_partkey), (part, p_name), (part, p_mfgr), (part, p_brand),
(part, p_type), (part, p_size), (part, p_container), (part, p_retailprice),
(part, p_comment), (supplier, s_suppkey), (supplier, s_name), (supplier,
s_address), (supplier, s_nationkey), (supplier, s_phone), (supplier, s_acctbal),
(supplier, s_comment), (partsupp, ps_partkey), (partsupp, ps_suppkey),
(partsupp, ps_availqty), (partsupp, ps_supplycost), (partsupp, ps_comment),
(customer, c_custkey), (customer, c_name), (customer, c_address), (customer,
c_nationkey), (customer, c_phone), (customer, c_acctbal), (customer,
c_mktsegment), (customer, c_comment), (nation, n_nationkey), (nation, n_name),
(nation, n_regionkey), (nation, n_comment), (lineitem, l_orderkey), (lineitem,
l_partkey), (lineitem, l_suppkey), (lineitem, l_linenumber), (lineitem,
l_quantity), (lineitem, l_extendedprice), (lineitem, l_discount), (lineitem,
l_tax), (lineitem, l_returnflag), (lineitem, l_linestatus), (lineitem,
l_shipdate), (lineitem, l_commitdate), (lineitem, l_receiptdate), (lineitem,
l_shipinstruct), (lineitem, l_shipmode), (lineitem, l_comment), (region,
r_regionkey), (region, r_name), (region, r_comment), (orders, o_orderkey),
(orders, o_custkey), (orders, o_orderstatus), (orders, o_totalprice), (orders,
o_orderdate), (orders, o_orderpriority), (orders, o_clerk), (orders,
o_shippriority), (orders, o_comment)]

[61 rows x 0 columns]
```

The above dataframe is now in the format where the cardinality estimates can be added as columns. Before the actual values are determined, placeholder values of 0 will be inserted into the dataframe.

[13]:
```
relation_df[['T(R)', 'V(R, A)', 'T(S) = T(R) / V(R, A)']] = 0
```

[14]:
```
relation_df
```

[14]:

Relation	Attribute	T(R)	V(R, A)	T(S) = T(R) / V(R, A)
part	p_partkey	0	0	0
	p_name	0	0	0
	p_mfgr	0	0	0
	p_brand	0	0	0
	p_type	0	0	0
…	…	…	…	…
orders	o_orderdate	0	0	0
	o_orderpriority	0	0	0
	o_clerk	0	0	0
	o_shippriority	0	0	0
	o_comment	0	0	0

[61 rows x 3 columns]

The first column to be updated will be T(R), or the number of tuples in each relation.

[15]:
```
def query_n_tuples(relation):

    result = []

    query = f'select count(*) from {relation}'

    cursor.execute(query)

    for row in cursor.fetchall():

        result.append(row)

    return result[0][0]
```

[16]:
```
for relation in relations.keys():

    index_condition = relation_df.index.get_level_values('Relation') == relation

    relation_df.loc[index_condition, 'T(R)'] = relation_df.loc[index_condition,␣
 ↪'T(R)'].apply(
        lambda x: query_n_tuples(relation))
```

[17]:
```
relation_df
```

[17]:

Relation	Attribute	T(R)	V(R, A)	T(S) = T(R) / V(R, A)
part	p_partkey	200000	0	0
	p_name	200000	0	0

```
          p_mfgr            200000        0                    0
          p_brand           200000        0                    0
          p_type            200000        0                    0
…                             …         …                    …
orders    o_orderdate      1500000        0                    0
          o_orderpriority  1500000        0                    0
          o_clerk          1500000        0                    0
          o_shippriority   1500000        0                    0
          o_comment        1500000        0                    0

[61 rows x 3 columns]
```

Next, V(R, A) or the number of distinct values in each attribute will be computed.

[18]:
```
def query_distinct_values(relation, attribute):

    result = []

    query = f'select count(distinct({attribute})) from {relation}'

    cursor.execute(query)

    for row in cursor.fetchall():

        result.append(row)

    return result[0][0]
```

[19]:
```
for relation, attribute in zip(flattened_relations, flattened_attributes):

    index_condition = ((relation_df.index.get_level_values('Relation') ==␣
 ↪relation) &
                       (relation_df.index.get_level_values('Attribute') ==␣
 ↪attribute))

    relation_df.loc[index_condition, 'V(R, A)'] = relation_df.
 ↪loc[index_condition, 'V(R, A)'].apply(
        lambda x: query_distinct_values(relation, attribute))
```

[20]:
```
relation_df
```

[20]:
```
                           T(R)  V(R, A)  T(S) = T(R) / V(R, A)
Relation Attribute
part     p_partkey       200000   200000                    0
         p_name          200000   199997                    0
         p_mfgr          200000        5                    0
         p_brand         200000       25                    0
```

```
          p_type            200000      150                      0
…                             …       …                        …
orders    o_orderdate      1500000     2406                      0
          o_orderpriority  1500000        5                      0
          o_clerk          1500000     1000                      0
          o_shippriority   1500000        1                      0
          o_comment        1500000  1482071                      0

[61 rows x 3 columns]
```

Now, T(S) = T(R) / V(R, A) can be computed directly by dividing the two existing columns.

[21]:
```
relation_df['T(S) = T(R) / V(R, A)'] = relation_df['T(R)'] / relation_df['V(R,␣
↪A)']
```

[22]:
```
relation_df
```

[22]:
```
                              T(R)  V(R, A)  T(S) = T(R) / V(R, A)
Relation Attribute
part     p_partkey          200000   200000           1.000000e+00
         p_name             200000   199997           1.000015e+00
         p_mfgr             200000        5           4.000000e+04
         p_brand            200000       25           8.000000e+03
         p_type             200000      150           1.333333e+03
…                             …       …                        …
orders   o_orderdate       1500000     2406           6.234414e+02
         o_orderpriority   1500000        5           3.000000e+05
         o_clerk           1500000     1000           1.500000e+03
         o_shippriority    1500000        1           1.500000e+06
         o_comment         1500000  1482071           1.012097e+00

[61 rows x 3 columns]
```

For clarity, this column will be converted from floats to integers.

[23]:
```
relation_df['T(S) = T(R) / V(R, A)'] = relation_df['T(S) = T(R) / V(R, A)'].
↪apply(lambda x: int(x))
```

[24]:
```
relation_df
```

[24]:
```
                              T(R)  V(R, A)  T(S) = T(R) / V(R, A)
Relation Attribute
part     p_partkey          200000   200000                     1
         p_name             200000   199997                     1
         p_mfgr             200000        5                 40000
         p_brand            200000       25                  8000
         p_type             200000      150                  1333
…                             …       …                        …
```

```
orders    o_orderdate       1500000     2406                  623
          o_orderpriority   1500000        5                300000
          o_clerk           1500000     1000                 1500
          o_shippriority    1500000        1              1500000
          o_comment         1500000  1482071                    1

[61 rows x 3 columns]
```

[25]:
```
relation_df.to_excel('Cardinality Estimations.xlsx')
```

[26]:
```
cursor.close()
```

[27]:
```
conn.close()
```

Relation	Attribute	T(R)	V(R, A)	T(S) = T(R) / V(R, A)
part	p_partkey	200000	200000	1
	p_name	200000	199997	1
	p_mfgr	200000	5	40000
	p_brand	200000	25	8000
	p_type	200000	150	1333
	p_size	200000	50	4000
	p_container	200000	40	5000
	p_retailprice	200000	20899	9
	p_comment	200000	131753	1
supplier	s_suppkey	10000	10000	1
	s_name	10000	10000	1
	s_address	10000	10000	1
	s_nationkey	10000	25	400
	s_phone	10000	10000	1
	s_acctbal	10000	9955	1
	s_comment	10000	10000	1
partsupp	ps_partkey	800000	200000	4
	ps_suppkey	800000	10000	80
	ps_availqty	800000	9999	80
	ps_supplycost	800000	99865	8
	ps_comment	800000	799124	1
customer	c_custkey	150000	150000	1
	c_name	150000	150000	1
	c_address	150000	150000	1
	c_nationkey	150000	25	6000
	c_phone	150000	150000	1
	c_acctbal	150000	140187	1
	c_mktsegment	150000	5	30000
	c_comment	150000	149968	1
nation	n_nationkey	25	25	1
	n_name	25	25	1
	n_regionkey	25	5	5
	n_comment	25	25	1
lineitem	l_orderkey	6001215	1500000	4
	l_partkey	6001215	200000	30
	l_suppkey	6001215	10000	600
	l_linenumber	6001215	7	857316
	l_quantity	6001215	50	120024
	l_extendedprice	6001215	933900	6
	l_discount	6001215	11	545565
	l_tax	6001215	9	666801
	l_returnflag	6001215	3	2000405
	l_linestatus	6001215	2	3000607
	l_shipdate	6001215	2526	2375
	l_commitdate	6001215	2466	2433
	l_receiptdate	6001215	2554	2349
	l_shipinstruct	6001215	4	1500303
	l_shipmode	6001215	7	857316
	l_comment	6001215	4580667	1
region	r_regionkey	5	5	1
	r_name	5	5	1
	r_comment	5	5	1
orders	o_orderkey	1500000	1500000	1
	o_custkey	1500000	99996	15
	o_orderstatus	1500000	3	500000
	o_totalprice	1500000	1464556	1
	o_orderdate	1500000	2406	623
	o_orderpriority	1500000	5	300000
	o_clerk	1500000	1000	1500
	o_shippriority	1500000	1	1500000
	o_comment	1500000	1482071	1

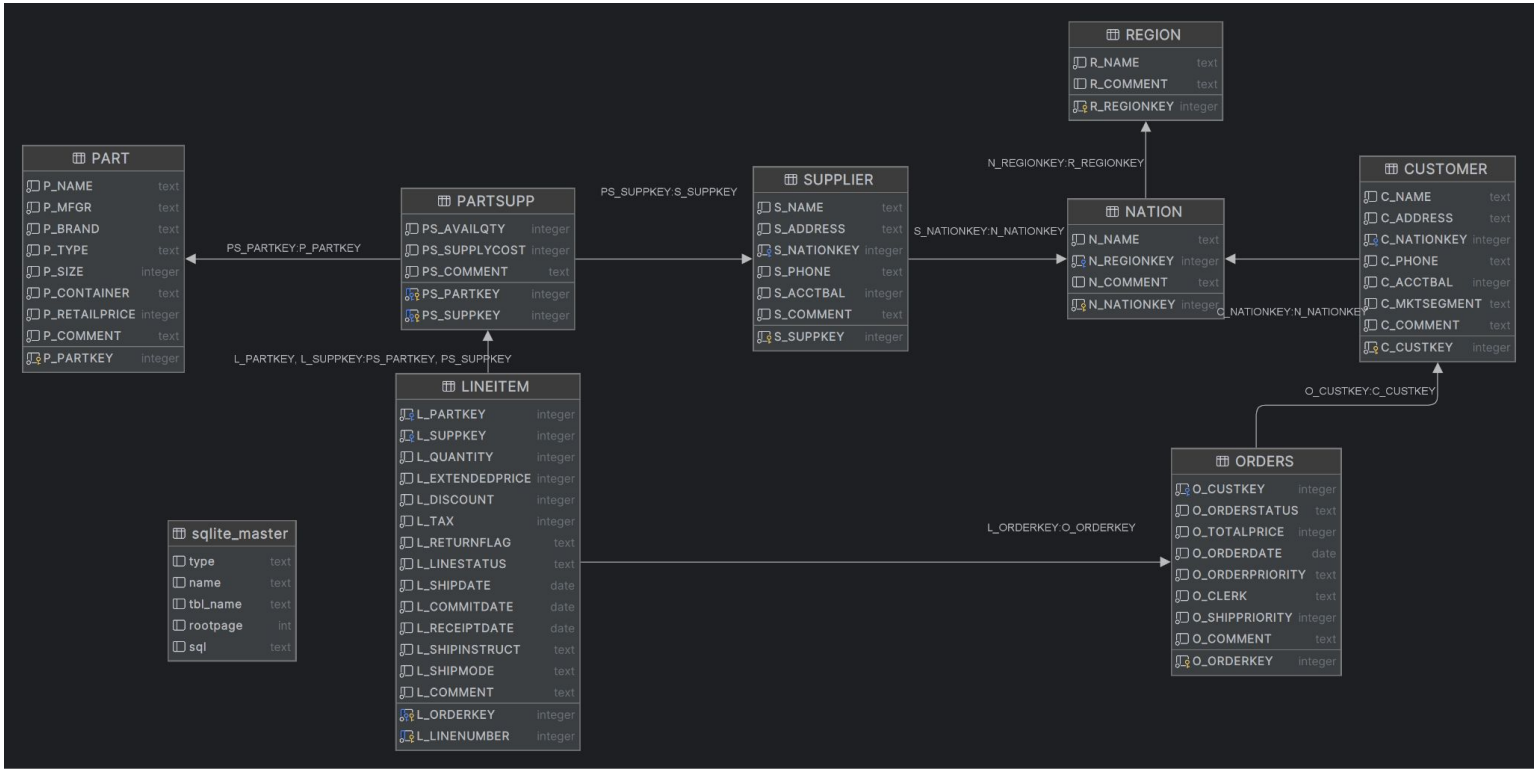
VLOOKUP TABLE	
Join Estimator - Enter relations and join "on" attributes	
Relation 1	**Relation 2**
orders	customer
"on" attribute	**"on" attribute**
ps_suppkey	l_suppkey
T(R1)	**T(R2)**
1500000	150000
V(R1, Y)	**V(R2, Y)**
10000	10000
Estimated Size	
22500000	

MANUAL ENTRY	
Join Estimator - Enter relations and join "on" attributes	
Relation 1	**Relation 2**
supplier	nation
"on" attribute	**"on" attribute**
s_nationkey	n_nationkey
T(R1)	**T(R2)**
10000	25
V(R1, Y)	**V(R2, Y)**
25	25
Estimated Size	
10000	

CS 542 Project
TPC-H Benchmark Analysis

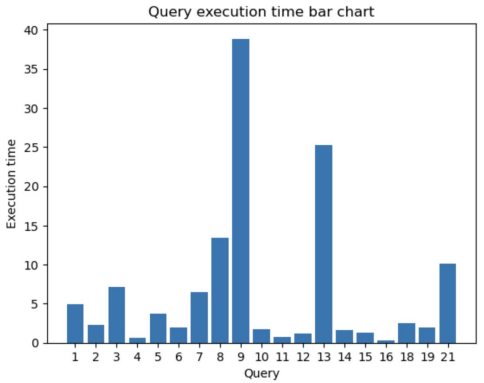
Group 5
Olivia Raisbeck, Phong Cao, Tyler Nardone

Entity-Relationship Diagram

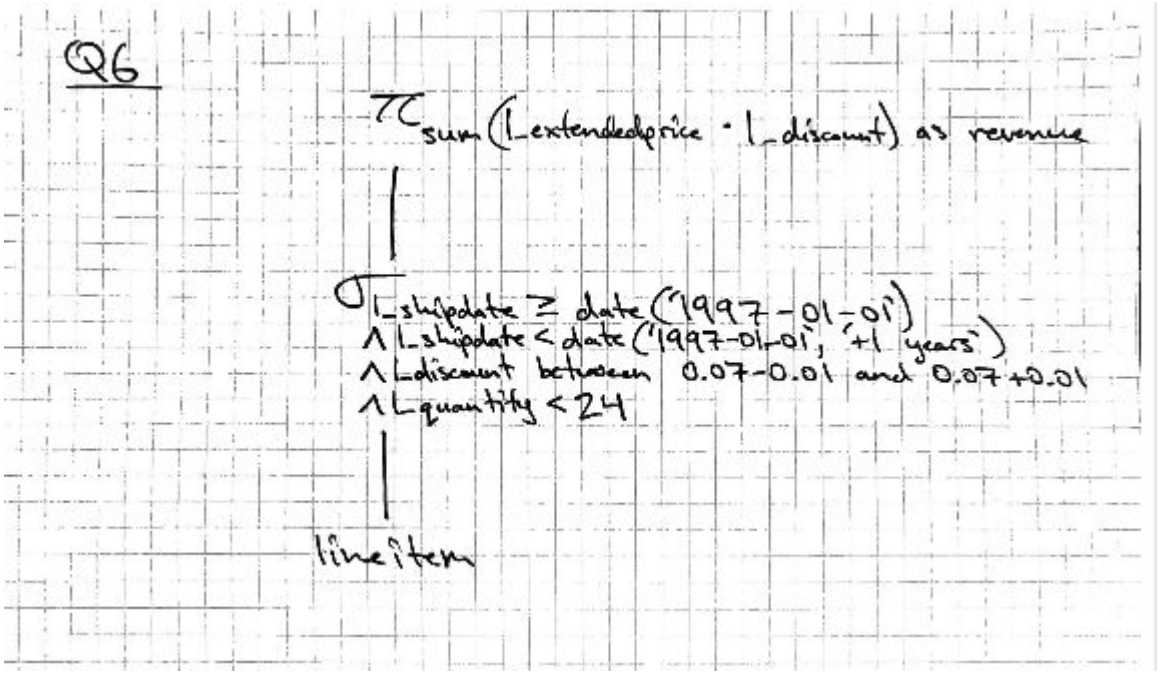


Query Execution Times

	Query	Execute time (s)
0	1	4.966
1	2	2.306
2	3	7.2
3	4	0.598
4	5	3.702
5	6	1.98
6	7	6.505
7	8	13.483
8	9	38.871
9	10	1.767
10	11	0.738
11	12	1.229
12	13	25.226
13	14	1.634
14	15	1.247
15	16	0.288
16	17	4933.62
17	18	2.528
18	19	1.92
19	20	6508.47
20	21	10.132
21	22	683.172

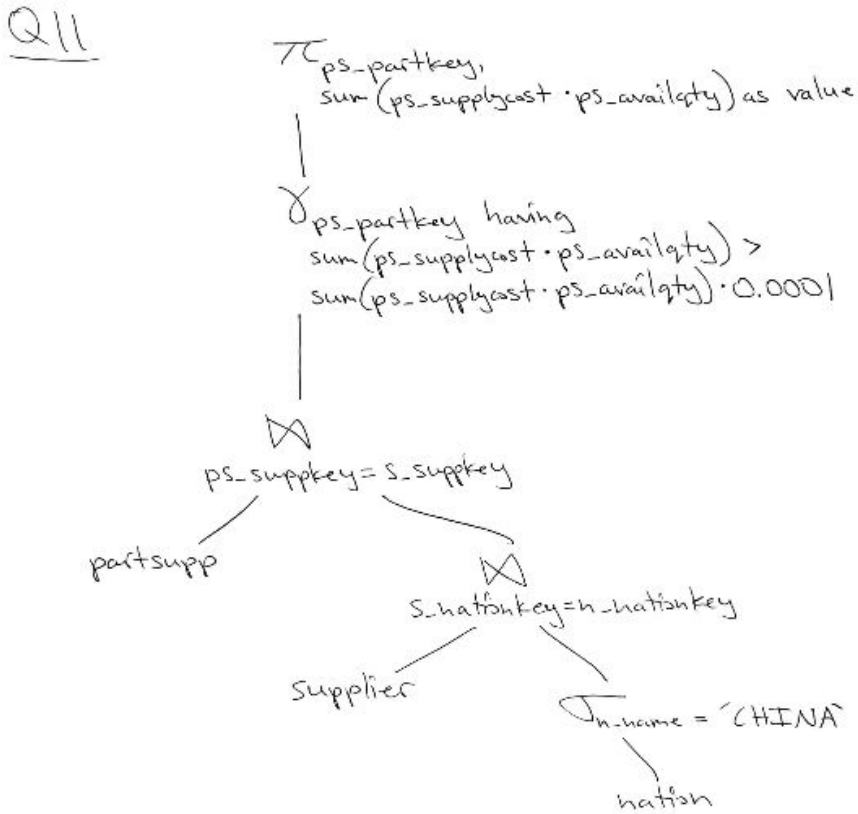


Relational Algebra Trees




```
        revenue
0  1.031663e+08
Size of query:  (1, 1)
```

Relational Algebra Trees



	PS_PARTKEY	value
0	26964	19453391.03
1	181683	17650015.20
2	159774	17150761.56
3	181556	16128801.21
4	164951	15811680.50

Size of query: (869, 2)

Cardinality Estimations

Relation	Attribute	T(R)	V(R, A)	T(S) = T(R) / V(R, A)
part	p_partkey	200000	200000	1
	p_name	200000	199997	1
	p_mfgr	200000	5	40000
	p_brand	200000	25	8000
	p_type	200000	150	1333
	p_size	200000	50	4000
	p_container	200000	40	5000
	p_retailprice	200000	20899	9
	p_comment	200000	131753	1
supplier	s_suppkey	10000	10000	1
	s_name	10000	10000	1
	s_address	10000	10000	1
	s_nationkey	10000	25	400
	s_phone	10000	10000	1
	s_acctbal	10000	9955	1
	s_comment	10000	10000	1
partsupp	ps_partkey	800000	200000	4
	ps_suppkey	800000	10000	80
	ps_availqty	800000	9999	80
	ps_supplycost	800000	99865	8
	ps_comment	800000	799124	1
customer	c_custkey	150000	150000	1
	c_name	150000	150000	1
	c_address	150000	150000	1
	c_nationkey	150000	25	6000
	c_phone	150000	150000	1
	c_acctbal	150000	140187	1
	c_mktsegment	150000	5	30000
	c_comment	150000	149968	1

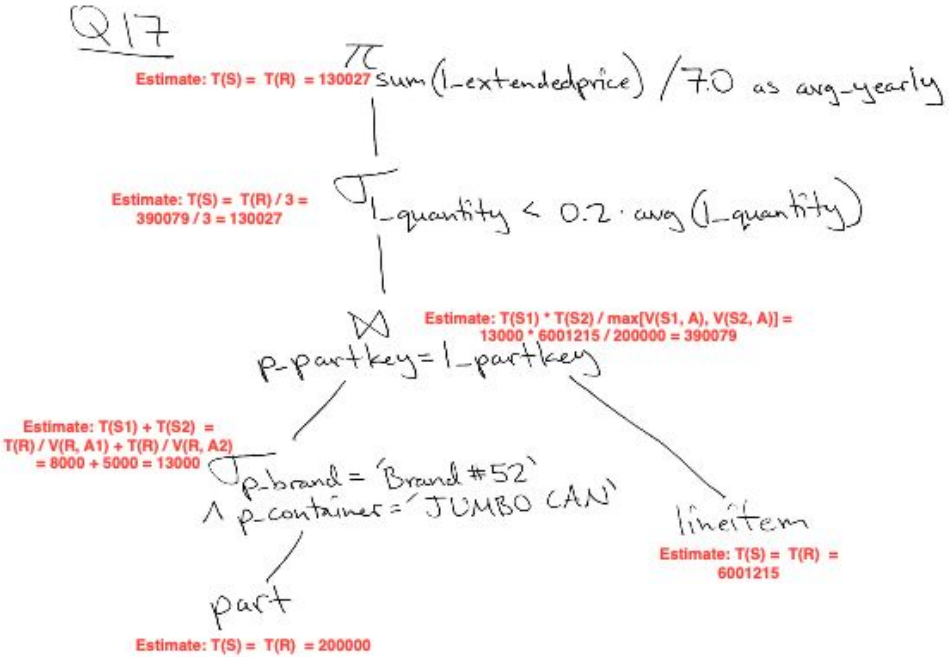
VLOOKUP TABLE	
Join Estimator - Enter relations and join "on" attributes	
Relation 1	Relation 2
orders	customer
"on" attribute	"on" attribute
ps_suppkey	l_suppkey
T(R1)	T(R2)
1500000	150000
V(R1, Y)	V(R2, Y)
10000	10000
Estimated Size	
22500000	

MANUAL ENTRY	
Join Estimator - Enter relations and join "on" attributes	
Relation 1	Relation 2
supplier	nation
"on" attribute	"on" attribute
s_nationkey	n_nationkey
T(R1)	T(R2)
10000	25
V(R1, Y)	V(R2, Y)
25	25
Estimated Size	
10000	

Cardinality Estimations

Query 17 Runtime:

4933.62 seconds

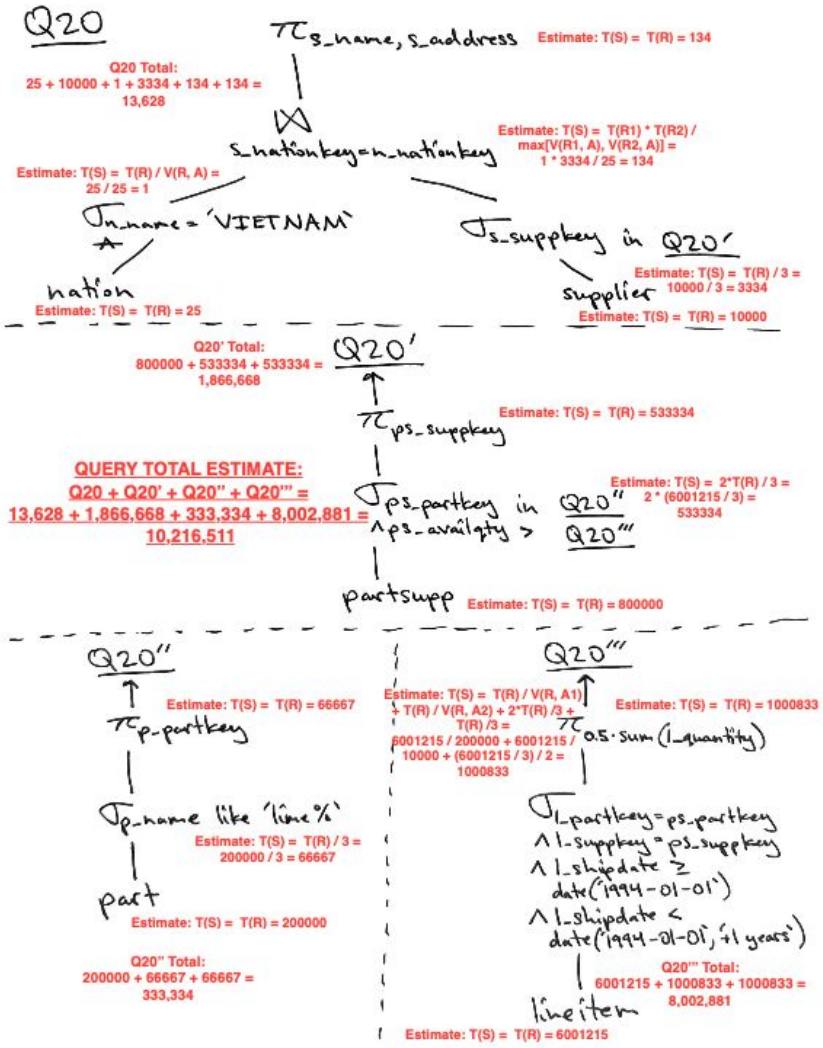


Estimated Query Total: 13000 + 200000 + 6001215 + 390079 + 130027 + 130027 = 6,864,348

Cardinality Estimations

Query 20 Runtime:

6508.473 seconds

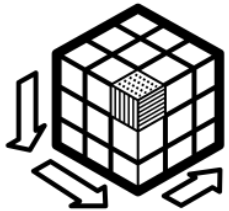


Cardinality Estimations

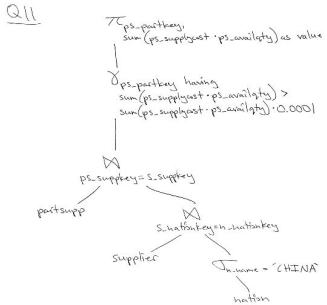
Cardinality estimations produced a high degree of variance among the number of tuples processed per second of runtime.

Query	Runtime	Cardinality Estimate	Tuples/Second
Q8	38.871	2,667	68
Q13	25.226	1,500,000	59,462
Q17	4933.62	6,864,348	1,391
Q20	6508.473	10,216,511	1,570
Q21	10.132	80,016	7,897
Q22	683.172	3,133,334	4,586

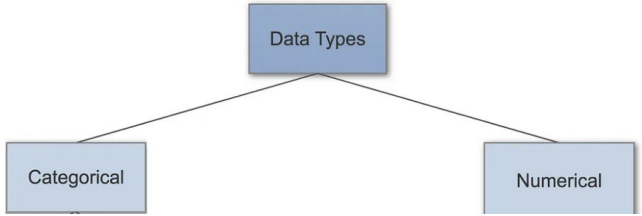
Factors that affect the execution time



Dimensional of query



Query structure



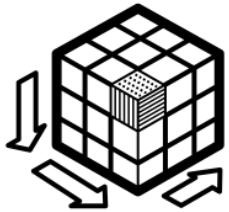
Using sub-query as a condition

Aggregation

Group by

Database size

Optimization idea



Reduce dimension of query

Merge table and then use selection

Reduce the workload of query

Push the projections down

Use selection before merge tables

Example:

Q20:

```
select
        s_name,
        s_address
from
        supplier,
        nation
where
        s_suppkey in (
                select
                        ps_suppkey
                from
                        partsupp
                where
                        ps_partkey in (
                                select
                                        p_partkey
                                from
                                        part
                                where
                                        p_name like 'lime%'
                        )
                        and ps_availqty > (
                                select
                                        0.5 * sum(l_quantity)
                                from
                                        lineitem
                                where
                                        l_partkey = ps_partkey
                                        and l_suppkey = ps_suppkey
                                        and l_shipdate >= date ('1994-01-01')
                                        and l_shipdate < date ('1994-01-01',
'+1 years')
                        )
        )
        and s_nationkey = n_nationkey
        and n_name = 'VIETNAM'
order by
        s_name;
```

Conclusions

- The 22 queries display a high degree of variance in the required runtime.
- Various opportunities for optimization have been identified.
- Relational algebra trees/cardinality estimations provided valuable insights into the query structures, partly explain the high runtimes of certain queries.
- Giving idea about the factors that affect to the query execution time
- Giving optimization idea