EXAM 2

Octavio Villalaz

1)

a)

***SELECT COUNT(\*) FROM scale1.customer;***

***result = 150000***

***SELECT COUNT(\*) FROM scale1.lineitem;***

***result = 6001215***

***SELECT COUNT(\*) FROM scale1.orders;***

***result = 1500000***

***SELECT COUNT(\*) FROM scale1.part;***

***result = 200000***

***SELECT COUNT(\*) FROM scale1.partsupp;***

***result = 800000***

***SELECT COUNT(\*) FROM scale1.region;***

***result = 5***

***SELECT COUNT(\*) FROM scale1.supplier;***

***result = 10000***

***SELECT COUNT(\*) FROM scale1.nation;***

***result = 25***

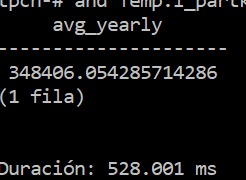
b)

A screenshot of a computer

Description automatically generatedDuration:

***12007464.963ms (3:20:07.465)***

c)

******Duration:

***528.001ms***

Unnested Query:

***CREATE TABLE scale1.Temp(l\_partkey,avg) as***

***SELECT l\_partkey, 0.2 \* avg(l\_quantity)***

***FROM scale1.lineitem***

***GROUP BY l\_partkey;***

***SELECT***

***sum(l\_extendedprice) / 7.0 as avg\_yearly***

***FROM***

***scale1.lineitem as l,***

***scale1.part,***

***scale1.Temp***

***WHERE***

***p\_partkey = l.l\_partkey***

***and p\_brand = 'Brand#23'***

***and p\_container = 'MED BOX'***

***and l\_quantity < Temp.avg***

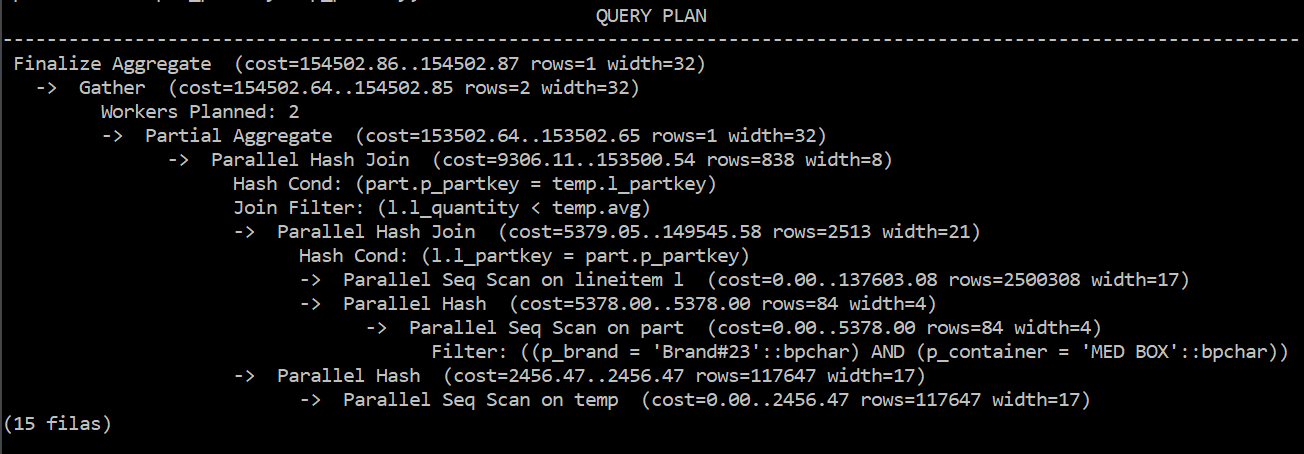
***and Temp.l\_partkey = p\_partkey;***

d)

A computer screen shot of a computer program

Description automatically generated

Normal Q17 Query



Unnested Q17 Query

The overall estimated cost of the unnested Q17 query plan (154502.87) is lower compared to the normal Q17 query plan (2070858.42). This cost reductions made the unnested Q17 query take less than a second to run compared to the 3 hours and 20 min for the normal Q17 query. Some key differences that make the unnested Q17 query plan faster are:

* Parallel Hash Join which distribute the join operation across multiple workers.
* Partial Aggregate step before finalizing the aggregate result. This is more efficient than computing the aggregate over the entire dataset in one go.

2)

a)

***CREATE MATERIALIZED VIEW scale1.mv\_q5 AS***

***select***

***l\_suppkey,***

***c\_nationkey,***

***sum(l\_extendedprice \* (1 - l\_discount)) as revenue***

***from***

***scale1.orders,***

***scale1.customer,***

***scale1.lineitem***

***where***

***o\_custkey = c\_custkey***

***and l\_orderkey = o\_orderkey***

***and o\_orderdate >= date '1994-01-01'***

***and o\_orderdate < date '1994-01-01' + interval '1' year***

***group by***

***l\_suppkey, c\_nationkey;***

b)

Original TPCH Q5 Benchmark query

A screenshot of a computer

Description automatically generated

***time = 509.600ms***

c)

***select***

***n\_name,***

***SUM(revenue) as revenue***

***from***

***scale1.mv\_q5,***

***scale1.supplier,***

***scale1.nation,***

***scale1.region***

***where***

***l\_suppkey = s\_suppkey***

***and c\_nationkey = s\_nationkey***

***and s\_nationkey = n\_nationkey***

***and n\_regionkey = r\_regionkey***

***and r\_name = 'ASIA'***

***group by***

***n\_name***

***order by***

***revenue desc;***

d) Query using materialized view

A screenshot of a computer

Description automatically generated

***Time = 53.180 ms***

3)

Insertion Tables

***CREATE TABLE ORDER\_INSERTIONS (***

***o\_orderkey INT PRIMARY KEY,***

***c\_custkey INT***

***);***

***CREATE TABLE CUSTOMER\_INSERTIONS (***

***c\_custkey INT PRIMARY KEY***

***);***

*Trigger for ORDER table*

***CREATE TRIGGER order\_insert\_trigger***

***AFTER INSERT ON ORDER***

***FOR EACH ROW***

***BEGIN***

***INSERT INTO ORDER\_INSERTIONS (o\_orderkey, c\_custkey)***

***VALUES (NEW.o\_orderkey, NEW.c\_custkey);***

***END;***

*Trigger for CUSTOMER table*

***CREATE TRIGGER customer\_insert\_trigger***

***AFTER INSERT ON CUSTOMER***

***FOR EACH ROW***

***BEGIN***

***INSERT INTO CUSTOMER\_INSERTIONS (c\_custkey)***

***VALUES (NEW.c\_custkey);***

***END;***

*Updater(Count Alogrithm)*

***UPDATE OrdCost***

***SET sum(o\_totalprice) = (***

***SELECT SUM(o\_totalprice)***

***FROM ORDER\_INSERTIONS oi***

***JOIN CUSTOMER\_INSERTIONS ci ON oi.c\_custkey = ci.c\_custkey***

***JOIN ORDER o ON oi.o\_orderkey = o.o\_orderkey***

***JOIN CUSTOMER c ON oi.c\_custkey = c.c\_custkey***

***WHERE o.o\_shippriority = 0 AND c.c\_acctbal > 0***

***);***

4)

See SQL Script “sakila-schema-modified-to-postgres”

5)

a)

L\_ORDERKEY: 4 bytes (integer)

L\_PARTKEY: 4 bytes (integer)

L\_SUPPKEY: 4 bytes (integer)

L\_LINENUMBER: 4 bytes (integer)

L\_QUANTITY: 8 bytes (decimal)

L\_EXTENDEDPRICE: 8 bytes (decimal)

L\_DISCOUNT: 8 bytes (decimal)

L\_TAX: 8 bytes (decimal)

L\_RETURNFLAG: 1 byte (char)

L\_LINESTATUS: 1 byte (char)

L\_SHIPDATE: 4 bytes (date)

L\_COMMITDATE: 4 bytes (date)

L\_RECEIPTDATE: 4 bytes (date)

L\_SHIPINSTRUCT: 25 bytes (char)

L\_SHIPMODE: 10 bytes (char)

L\_COMMENT: 44 bytes (varchar)

Adding these up gives us a total row size of 124 bytes.

So, the size of storing the Lineitem table in a traditional (row-based) file would be approximately:

6001215 \* 124 bytes = 744,148,860 bytes = ***744.15 megabytes***.

b)

Used columns:

L\_RETURNFLAG: 1 byte (char) / 5 = 0.2 bytes

L\_LINESTATUS: 1 byte (char) / 5 = 0.2 bytes

L\_QUANTITY: 8 bytes / 5 = 1.6 bytes

L\_EXTENDEDPRICE: 8 bytes / 5 = 1.6 bytes

L\_DISCOUNT: 8 bytes / 5 = 1.6 bytes

L\_TAX: 8 bytes / 5 = 1.6 bytes

L\_SHIPDATE: 4 bytes (date) / 5 = 0.8 bytes

Total = 6001215 \* 7.6 bytes = 45,609,234 bytes = ***45.61 megabytes***.

c)

A computer screen shot of a black screen

Description automatically generated

***Cost = 231,176.37***

d)

Everything will be the same except the last operation ”Parallel Seq Scan on lineitem” that is replaced with:

***Late Materialized Columnar Scan on lineitem***

***Filter: (l\_shipdate <= '1998-09-02 00:00:00'::timestamp without time zone)***

e)

M = 124 bytes

N = 7.6 bytes

Cost = (124log(124) ) + (7.6log(7.6)) + 124 + 7.6 = ***744.288***

f)

Looking at the cost of the columnar database we can see that it has a much lower cost meaning that doing a merge-sort join will be faster.