***به یاد خداوند عشق و آزادی***

**گزارش پروژه پایگاه داده پیشرفته**

نام و نام خانوادگی : محمد مهدی زارعیان

شماره دانشجویی : 97112092

...................................................................................................

**تعهد نامه :**

اینجانب محمد مهدی زارعیان به شماره دانشجویی 97112092 متعهد می شوم که در این پروژه مربوط به درس پایگاه داده پیشرفته را از فرد یا افراد یا سایت یامکانی دیگر کپی برداری نکرده و حاصل زحمت و وقت گذاشتن شخص اینجانب می باشد و تنها در مواردی از همکاری و کمک دیگر دوستان استفاده کرده ام .

همچنین میپذیرم که در صورت اثبات کپی برداری این پروژه از جایی و یا اینکه خود مسئول و انجام دهنده آن نبوده باشم ، تبعات آن که توسط استاد تعیین می شود را بپذیرم .

....................................................................................................

**شرح گزارش :**

در این گزارش به صورت مرحله به مرحله کارها و فعالیت های صورت گرفته در مسیر پروژه ذکر خواهد شد و در انتها نیز جمع بندی و مقایسه هر یک از 22 کوئری روی 4 حالت اجرا شده یعنی postgresql و Spark + Parquet + HDFS و

Spark + ORC + HDFS و Spark + Avro + HDFS به صورت یکجا به ازای هر کوئری منتشر خواهد شد . در ضمن هر جایی که کد و یا مطلبی استفاده شده باشد عینا در فایل ضمیمه با شماره ای که در متن کنار آن ذکر خواهد شد ، با همان شماره در فایل های ضمیمه شده به ترتیب موجود است . قسمتی از کد ها نیز در حین ارایه گزارش دراینجا ذکر شده اند که آنها نیز به طور کامل در پیوست موجود است . همچنین در هر جایی از پروژه که به هر دلیلی کوئری ها قابل اجرا نبود و من نتوانستم ایراد آنها را رفع کنم در مرحله مقایسه زمان ها با حروف قرمز رنگ مشخص شده اند .

\*قسمت POSTGRESQL \*

در ابتدا پس انجام کارهای اولیه مانند ساختن کلید خصوصی با نرم افزار putty و وارد شدن به سرور و بعد از آن داکر مربوطه می بایست در داکر postgres خود جدولا را بسازیم . این جدولا در پیوست با شماره (1) موجود اند . به عنوان مثال برای ساختن جدول Lineitem که بزرگترین جدول نیز هست از کد زیر استفاده کرده ایم :

DROP TABLE IF EXISTS LINEITEM CASCADE;

CREATE TABLE LINEITEM (

L\_ORDERKEY BIGINT NOT NULL, -- references O\_ORDERKEY

L\_PARTKEY BIGINT NOT NULL, -- references P\_PARTKEY (compound fk to PARTSUPP)

L\_SUPPKEY BIGINT NOT NULL, -- references S\_SUPPKEY (compound fk to PARTSUPP)

L\_LINENUMBER INTEGER,

L\_QUANTITY DECIMAL,

L\_EXTENDEDPRICE DECIMAL,

L\_DISCOUNT DECIMAL,

L\_TAX DECIMAL,

L\_RETURNFLAG CHAR(1),

L\_LINESTATUS CHAR(1),

L\_SHIPDATE DATE,

L\_COMMITDATE DATE,

L\_RECEIPTDATE DATE,

L\_SHIPINSTRUCT CHAR(25),

L\_SHIPMODE CHAR(10),

L\_COMMENT VARCHAR(44),

PRIMARY KEY (L\_ORDERKEY, L\_LINENUMBER),

TEMP VARCHAR(10)

);

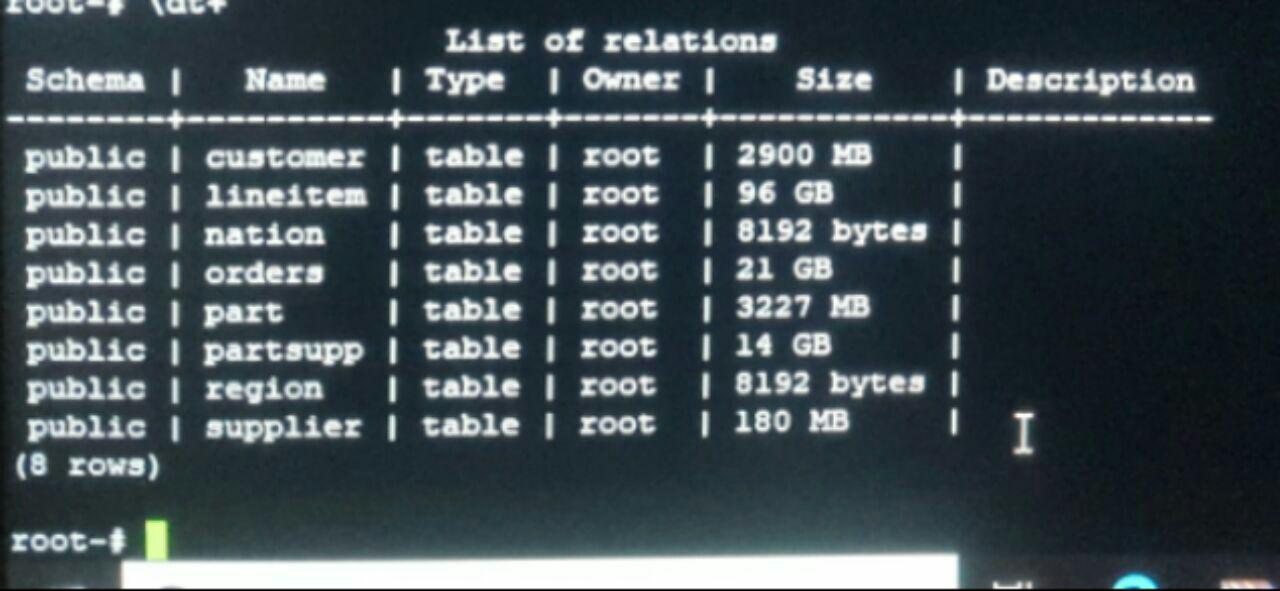
همانطور که ذکر شد کد مابقی جداول در پیوست (1) موجود است .

در ادامه می بایست حدود 170 گیگ دیتاهای جداول را در 8 جدول موجود بریزیم یا به عبارتی دقیقتر از سرور کپی کنیم تا جداول برای هرفرد از روی جداول روی سرور اصلی کپی صورت بگیرد . این کار و به راحتی با csv کردن به راحتی به صورت

\copy {table\_name} FROM '{table\_name}.csv' (FORMAT CSV, DELIMITER '|');

همچنین در اینجا ما یک ستون کاذب در انتهای هر جدول ایجاد میکنیم که پس از ایجاد جدول آن را drop میکنیم . دلیل این کار این است که بدون این افزودن ستون برای خوندن اخرین ستون به مشکل بر میخوردیم وبعضا جداول داده های ستون آخر را از دست می دهند .

این عمل ریختن داده ها در جدول عملی زمانبر است و به عنوان مثال برای بزرگترین جدول Lineitem که حدود 96 گیگ است حدود 17 ساعت زمان برد ! برای اینکه در حین انجام کارها چه دراین مرحله و چه در مرحله ران کردن کوئری ها در صورت قطع اینترنت کار متوقف نشود و یا اینکه نیازی به روشن بودن سیستم در حین انجام عملیات نباشد ، از screen ها در لینوکس استفاده کردم که در اینجا استفاده از انها به دلیل ذکر شده در بالا اجتناب ناپذیر بود .حجم جداول به صورت زیر است که با دستور \dt+ نشان داده میشود .



سپس می بایست تک تک کوئری ها را اجرا کنیم .

در گام نخست قبل از اجرا باید جاهایی در کوئری ها که با علامت دو نقطه + عدد وجود دارد که مقدار دهی کنیم . مقداری که فقط کافیست معتبر باشد . من برای این کار مثلا وقتی که مقداری رو باید از جدول nation جایگزین کنیم با روش زیر 10 تای اول اون جدولو رو میدیدم و سپس مقداری معتبر رو از روی اونها به دست میاوردم . مثلا اگر تاریخ ارسال محصولی رو میخواستیم با این کار یه حدود تاریخ معتبری به دست می آمد ودر هر کوئری جایگزین میکردم . بدین صورت 22 تا کوئری به نحوی که در پیوست (2) وجود دارد ، به دست می آید . حال باید این 22 تاکوئری را اجرا و زمان پردازش که در واقع عملا فقط صرف disk i/o میشود را گزارش کنیم زیرا در postgresql از پردازش موازی و خوشه ای بهره نمیبریم و گویا کل کار تنها روی یک CPU درحال انجام است . به طبع هم برای اجرا همانگونه که در قبل ذکر شد مجبور به استفاده از screen ها هستیم چون همانگونه که در پیوست نیز میتوان دید گاهی اجرای یک کوئری 15 الی 20 ساعت زمان میبرد !

برای انکه خروجی زمان و نحوه اجرای کوئری ( پلن منطقی اجرای آن شامل ترتیب پیاده سازی join ها و Select ها توسط postgresql) را نیز ببینیم در ابتدا از explain analyze استفاده کردیم و به عنوان مثال برای اولین کوئری نیز از

\o 1.analize.out بهره میبریم که خروجی گزارش را در داکر ذخیره کنیم وبعدا بتوانیم در گزارش بیاوریم . خروجی explain analyze هر 22 تا کوئری در پیوست (3) آمده است . البته من کوئری 20 رو بعد از چند بار اجرا و دادن زمان 24 ساعت به هر اجرا موفق به خروجی گرفتن نشدم که نتونستم ایراد رو حل کنم !

به عنوان مثال در زیر خروجی explain analyze کوئری 9 آورده شده است :

--------------------------------------

QUERY PLAN

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Finalize GroupAggregate (cost=34309479.68..34728923.24 rows=409020 width=144) (actual time=8961020.176..8977610.995 rows=175 loops=1)

Group Key: nation.n\_name, (date\_part('year'::text, (orders.o\_orderdate)::timestamp without time zone))

-> Gather Merge (cost=34309479.68..34713584.99 rows=818040 width=144) (actual time=8960913.274..8987552.625 rows=525 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial GroupAggregate (cost=34308479.66..34618162.85 rows=409020 width=144) (actual time=8960371.001..8976846.977 rows=175 loops=3)

Group Key: nation.n\_name, (date\_part('year'::text, (orders.o\_orderdate)::timestamp without time zone))

-> Sort (cost=34308479.66..34346295.33 rows=15126267 width=135) (actual time=8960312.436..8964489.977 rows=10870910 loops=3)

Sort Key: nation.n\_name, (date\_part('year'::text, (orders.o\_orderdate)::timestamp without time zone)) DESC

Sort Method: external merge Disk: 807816kB

Worker 0: Sort Method: external merge Disk: 792176kB

Worker 1: Sort Method: external merge Disk: 807720kB

-> Hash Join (cost=7847955.33..29402563.02 rows=15126267 width=135) (actual time=8498483.957..8927655.655 rows=10870910 loops=3)

Hash Cond: (supplier.s\_nationkey = nation.n\_nationkey)

-> Parallel Hash Join (cost=7847941.51..29286221.53 rows=15126267 width=35) (actual time=8498448.484..8922000.545 rows=10870910 loops=3)

Hash Cond: (lineitem.l\_orderkey = orders.o\_orderkey)

-> Parallel Hash Join (cost=3421014.01..24339108.58 rows=15126267 width=39) (actual time=7770090.989..7776330.819 rows=10870910 loops=3)

Hash Cond: (lineitem.l\_suppkey = supplier.s\_suppkey)

-> Parallel Hash Join (cost=3386551.01..23997011.96 rows=15126267 width=47) (actual time=7130591.186..7739608.001 rows=10870910 loops=3)

Hash Cond: ((lineitem.l\_suppkey = partsupp.ps\_suppkey) AND (lineitem.l\_partkey = partsupp.ps\_partkey))

-> Parallel Hash Join (cost=525358.01..20595286.06 rows=15126267 width=45) (actual time=4932038.388..6254839.186 rows=10870910 loops=3)

Hash Cond: (lineitem.l\_partkey = part.p\_partkey)

-> Parallel Seq Scan on lineitem (cost=0.00..15025590.60 rows=249583360 width=41) (actual time=35.756..4500304.232 rows=200012634 loops=3)

-> Parallel Hash (cost=517072.66..517072.66 rows=504988 width=4) (actual time=247599.441..247599.445 rows=362304 loops=3)

Buckets: 131072 Batches: 16 Memory Usage: 3712kB

-> Parallel Seq Scan on part (cost=0.00..517072.66 rows=504988 width=4) (actual time=440.093..246638.457 rows=362304 loops=3)

Filter: ((p\_name)::text ~~ '%hot%'::text)

Rows Removed by Filter: 6304363

-> Parallel Hash (cost=2166182.40..2166182.40 rows=33318840 width=22) (actual time=822318.614..822318.614 rows=26666667 loops=3)

Buckets: 65536 Batches: 2048 Memory Usage: 2720kB

-> Parallel Seq Scan on partsupp (cost=0.00..2166182.40 rows=33318840 width=22) (actual time=11.876..789045.928 rows=26666667 loops=3)

-> Parallel Hash (cost=27219.67..27219.67 rows=416667 width=12) (actual time=12805.379..12805.379 rows=333333 loops=3)

Buckets: 131072 Batches: 16 Memory Usage: 4000kB

-> Parallel Seq Scan on supplier (cost=0.00..27219.67 rows=416667 width=12) (actual time=0.046..12597.959 rows=333333 loops=3)

-> Parallel Hash (cost=3401577.33..3401577.33 rows=62497533 width=8) (actual time=713791.589..713791.589 rows=50000000 loops=3)

Buckets: 131072 Batches: 2048 Memory Usage: 3936kB

-> Parallel Seq Scan on orders (cost=0.00..3401577.33 rows=62497533 width=8) (actual time=32.735..669941.442 rows=50000000 loops=3)

-> Hash (cost=11.70..11.70 rows=170 width=108) (actual time=24.610..24.610 rows=25 loops=3)

Buckets: 1024 Batches: 1 Memory Usage: 10kB

-> Seq Scan on nation (cost=0.00..11.70 rows=170 width=108) (actual time=24.584..24.592 rows=25 loops=3)

Planning Time: 1145.658 ms

Execution Time: 8988129.249 ms

(42 rows)

همانطور که در بالا نیز میبینید زمان اجرا و زمان برنامه ریزی (planning) در انتها آمده شده است و اطلاعات دیگری مانند سطرهای درگیر در هر خط و مموری مورد استفاده و ... آورده شده است .

در انتها نیز در جدول زیر زمان اجرای هر کوئری را مشاهده میکنیم :

|  |  |  |
| --- | --- | --- |
| Execution Time به دقیقه | Execution Time به ثانیه | شماره کوئری |
| 35 | 2085 | 1 |
| 2 | 103 | 2 |
| 65 | 3870 | 3 |
| 92 | 5488 | 4 |
| 113 | 6780 | 5 |
| 38 | 2286 | 6 |
| 76 | 4532 | 7 |
| 59 | 3498 | 8 |
| 150 | 8988 | 9 |
| 79 | 4693 | 10 |
| 14 | 825 | 11 |
| 274 | 16411 | 12 |
| 126 | 7548 | 13 |
| 295 | 17641 | 14 |
| 525 | 31460 | 15 |
| 57 | 3412 | 16 |
| 7 | 377 | 17 |
| 187 | 11207 | 18 |
| 1 ! | 28! | 19 |
| ناتوانی در اجرا | ناتوانی در اجرا | 20 |
| 215 | 12898 | 21 |
| 16 | 907 | 22 |

قبل از ورود به قسمت اصلی پروژه نیاز است شرح و مقایسه ای کلی میان روشهای قالب بندی دیتاها داشته باشیم . تا ببینیم انتظار چه تغییرات و اعمال و خروجی های متفاوتی در نهایت داریم . Avro و Parquet و ORC از قالب های نوین فایل هستند که می توانند در پروژه های کلان که حجم دیتاها بسیار زیاد است مفید باشند و در زیر به مختصر تعریف میشوند :

***Apache Avro***

این قالب فایل که جزء ساده ترین قالبهای موجود با محبوبیت زیاد است، یک قالب ستونی نیست بلکه تنها یک موتور فشرده سازی بسیار خوب و با قابلیت حفظ مدل داده های ماست. اگر بخواهیم دقیق تر صحبت کنیم، Avro یک پروتکل ذخیره باینری داده هاست (Data Serialization) که در اصل برای انتقال داده ابداع شده است اما می توان برای ذخیره داده ها هم از آن استفاده کرد.

اگر قصد دارید داده های خود را به صورت JSON در هدوپ ذخیره کنید که بعدها به راحتی بتوانید داده ها را ویرایش کرده و تغییر دهید، Avro مناسب کار شماست. البته در مقایسه با قالب های فایل ستونی، میزان فشرده سازی آن کمتر است اما قابلیت انعطاف پذیری زیادی را به شما هدیه می کند.

ضعف اصلی این قالب فایل هم، سطری بودن داده ها در آن است که برای تحلیل های داده که معمولاً روی یک ستون خاص تمرکز دارد، غیربهینه و کند است.

***Apache Parquet***

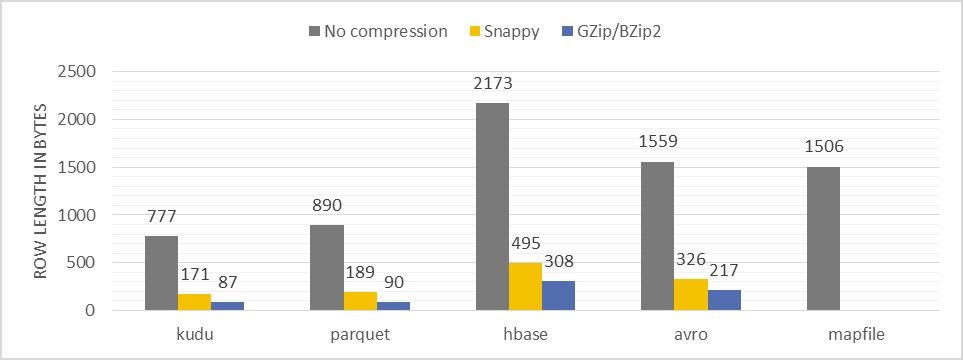
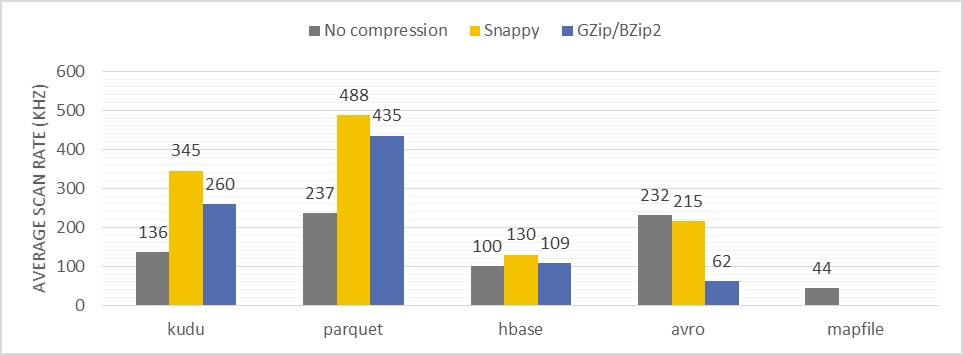
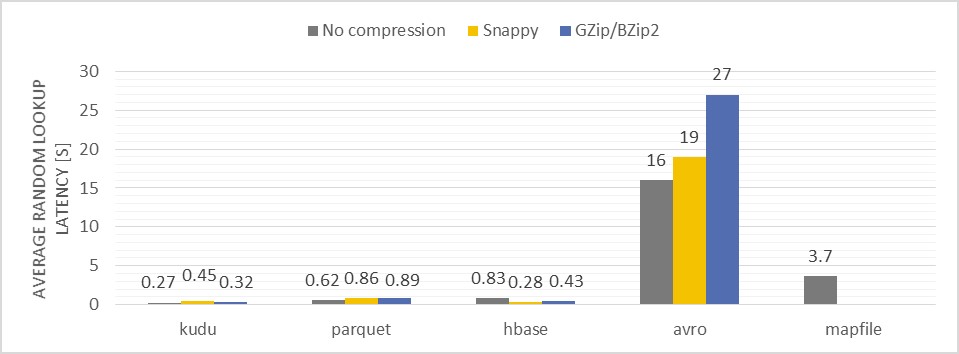
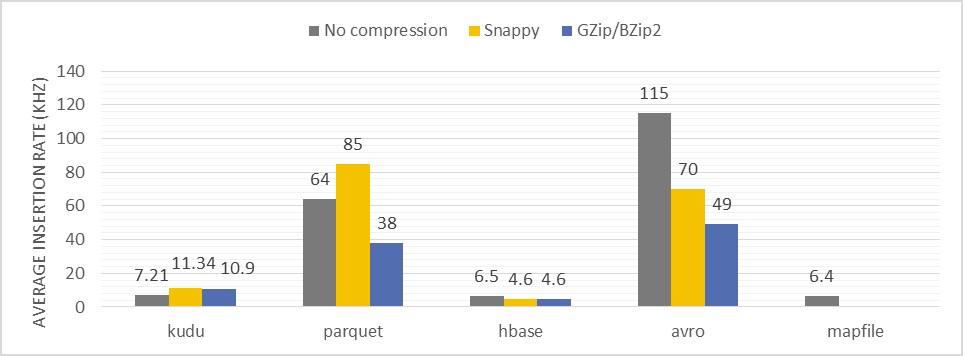
این قالب فایل بر خلاف Avro، با هدف افزایش سرعت جستجو و کاهش زمان مراجعه به دیسک طراحی شده است. Parquet که امروزه محبوب ترین و رایج ترین قالب ستونی ذخیره اطلاعات به حساب می آید، با نوع داده های مشخص و محدود خود، فرآیند فشرده سازی را تسهیل می کند و از طرفی برای افزایش بازدهی پرس و جوهای تحلیلی، اجازه ذخیره چندین نوع داده را در یک ستون به ما می دهد. مثلاً می توانید علاوه بر نام محصول، گروه و تاریخ فروش آنرا هم در یک ستون ذخیره کنید تا فیلتر کردن شما بر اساس گروه و بازه زمانی معین، بتواند به سرعت و بدون مراجعه به ستون های دیگر، قابل انجام باشد.

***Optimized Row Columnar (ORC)***

در سال ۲۰۱۳ برای افزایش بازدهی Hive که یکی از ابزار پرکاربرد حوزه کلان داده است، قالبی ستونی با قابلیت ایندکس زنی داده ها و نیز پشتیبانی از ACID (برای اجرای تراکنش ها ) ابداع شد با نام ORC که امروزه جزء پروژه های بنیاد Apache هم قرار گرفته است. این قالب فایل، مشابه پارکت (Parquet)، نوع داده های از پیش تعیین شده ای را برای ذخیره داده ها پیشنهاد می کند و بنابراین با انتخاب انکودینگ مناسب برای هر نوع داده،‌میزان فشرده سازی بسیار خوبی دارد . پشتیبانی از داده های جریانی و کش سمت کلاینت هم از دیگر مزیت های این قالب فایل است.

در مقایسه دو قالب ستونی ORC و پارکت ، تفاوت معناداری بین این دو گزارش نشده است و فقط برای کوئری هایی که فیلترهای مختلف را روی داده ها اعمال می کنند، سرعت پاسخگویی ORC به دلیل ساختار ایندکس گذاری مناسب که نیاز به خواندن کل فایل را از بین می برد، بیشتر ذکر شده است.

اکنون 4 نمودار زیر تا حدودی به مقایسه نسبی قالب های فایل های مختلف پرداخته اند:



در شمای کلی و به طور خلاصه می توان به جدول زیر نیز برای دیدن تفاوت ها دقت کرد :



در ادامه نیز انتظار داریم که این تفاوت ها در خروجی کوئری های ما نیز دیده شود . مثلا انتظار داریم تفاوت چندانی بین خروجی زمان Parquet و ORC نداشته باشیم .

................................................................................................

**قسمت Spark + parquet + HDFS :**

در این قسمت در ابتدا باید از روی جداول ساخته شده در مرحله قبل پستگره و داده های موجود در آنها ، عمل کپی با قالب فایل پارکت که بر روی HDFS ذخیره شود ، را انجام دهیم . به عبارت دقیقتر ما باید mapping انجام بدیم ، تا در این قسمت و دو قسمت بعدی اسپارک نیز بتوانیم از خاصیت پردازش موازی و ظرفیت سرور ها به درستی استفاده کنیم . در پیوست شماره (4) لیست کامل کد های تبدیل به پارکت آورده شده است . در اینجا به عنوان نمونه کد تبدیل برای جدول part را میبینیم :

from pyspark import SparkContext

from pyspark.sql import SQLContext

sc = SparkContext(appName="CSV2Parquet")

sqlContext = SQLContext(sc)

from pyspark.sql.types import \*

fields = [StructField("P\_PARTKEY", IntegerType(), False),

StructField("P\_NAME", StringType(), True),

StructField("P\_MFGR", StringType(), True),

StructField("P\_BRAND", StringType(), True),

StructField("P\_TYPE", StringType(), True),

StructField("P\_SIZE", IntegerType(), True),

StructField("P\_CONTAINER", StringType(), True),

StructField("P\_RETAILPRICE", FloatType(), True),

StructField("P\_COMMENT", StringType(), True)]

schema = StructType(fields)

rdd = sc.textFile("/data/OLAP\_Benchmark\_data/part.tbl")

part\_df = rdd.map(lambda x: x.split("|")) \

.map(lambda x: {'P\_PARTKEY': int(x[0]),

'P\_NAME': x[1],

'P\_MFGR': x[2],

'P\_BRAND': x[3],

'P\_TYPE': x[4],

'P\_SIZE': int(x[5]),

'P\_CONTAINER': x[6],

'P\_RETAILPRICE': float(x[7]),

'P\_COMMENT': x[8]}).toDF(schema)

part\_df.write.parquet("hdfs://namenode:8020/mahdi-parquet-data/part.parquet")

شاید در اینجا لازم باشد در ابتدا قبل از توضیح کد بالا به معرفی CSV بپردازیم . csv در واقع برای تبدیل یک مارتیس در یک فایل txt کاربرد دارد . ما اینجا جداول را به صورت csv در می آوریم . به این صورت که هر سطر ماتریس ( جدول ) یک خط در txt میشود و بین هر ستون در یک سطر نیز یک "," به عنوان جدا کننده ستون ها از هم قرار میدهیم . در csv افزودن یک سطر اول برای نام ستونها اختیاری می باشد .

همانطورکه در کد بالا نیز دیده میشود ابتدا با ورود به محیط pyspark در کانینر مربوطه به خود ،هر ستون جدول part را مثلا P\_BRAND را نوع آن را که عدد int باشد و یا رشته و یا نوع اعشاری باشد ، را تعیین میکنیم و در قسمت دوم که مپ کردن برای ساختن دیتافریم ( تقریبا معادل جدول در پوستگره ) انها را به این صورت مپ میکنیم . در انتها نیز مکانی که میخواهیم آن dataframe در انجا ذخیره شود را آدرس دهی میکنیم . این کار را برای تک تک 8 جدول تکرار میکنیم . اگر هم در زمانی در ساختن dataframe ها خطایی صورت دادیم با استفاده از دستور

hadoop fs -rm -R URI

مجددا فایل ساخته شده را پاک و دوباره تغییرات رو انجام داده و از اول شروع میکنیم.

حال نوبت به یک کار مهم می رسد و آن این است که باید تک تک کوئری ها را با پایتون تبدیل به نوعی قابل فهم برای spark کنیم . این کوئری ها را در هر 3 بخش اسپارک قابل اجرا هستند و هر بار فقط باید مانند مرحله قبل قالب ذخیره سازی را به طوری که در هر مورد در بخش مربوطه میبینیم ، تعویض کنیم . کار تبدیل کوئری ها کاری سخت و زمانگیر و جزو به جزو هست زیرا در pyspark از خیلی از خواص محیطی مانند postgresql خبری نیست و همچنین استاد اجازه استفاده از کدهای SQL را نیز به طبع نمیدهند زیرا عملا خاصیت پردازش موازی که در اسپارک داریم را بی استفاده میکند . البته میتوان کوئری ها را با scala نیز تبدیل کرد که من شناختی ازین زبان نداشتم ! در این قسمت من یک راهنما که با سرچ و جستوجهای موردی برای هر تبدیل پیدا کردم و انها را صرفا در یک فایل txt ذخیره کردم تا سرعتم برای تبدیل و برای رجوع مجدد را بالاببرم ، در پیوست شماره(5) آورده ام که شاید مفید باشد .

در نهایت کوئری های تبدیل شده در پیوست شماره (6) آورده شده اند .

به عنوان کوئری شماره 1 تبدیل شده به صورت زیر می باشد :

from decimal import \*

from pyspark.sql.types import \*

from pyspark.sql import functions as F

from pyspark.sql import \*

lineitem = sqlContext.read.parquet("hdfs://namenode:8020/mahdi-parquet-data/lineitem.parquet")

func1 = lambda x , y : x \* (1-y)

func2 = lambda x ,y , z : x \* (1-y) \* (1+z)

query1 = lineitem.filter( lineitem.L\_SHIPDATE < "1998-12-01")\

.select(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS , lineitem.L\_QUANTITY , lineitem.L\_EXTENDEDPRICE , lineitem.L\_DISCOUNT , lineitem.L\_TAX )\

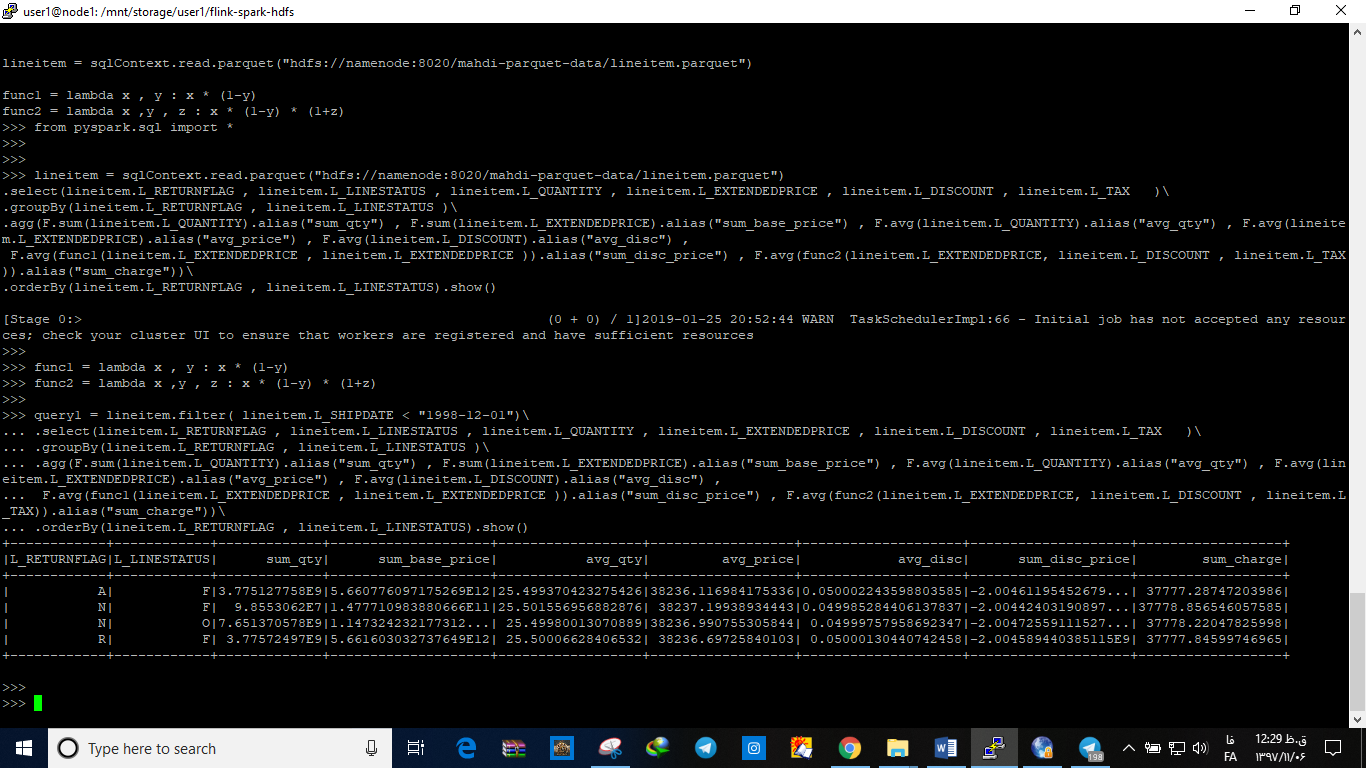
.groupBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS )\

.agg(F.sum(lineitem.L\_QUANTITY).alias("sum\_qty") , F.sum(lineitem.L\_EXTENDEDPRICE).alias("sum\_base\_price") , F.avg(lineitem.L\_QUANTITY).alias("avg\_qty") , F.avg(lineitem.L\_EXTENDEDPRICE).alias("avg\_price") , F.avg(lineitem.L\_DISCOUNT).alias("avg\_disc") ,

F.avg(func1(lineitem.L\_EXTENDEDPRICE , lineitem.L\_EXTENDEDPRICE )).alias("sum\_disc\_price") , F.avg(func2(lineitem.L\_EXTENDEDPRICE, lineitem.L\_DISCOUNT , lineitem.L\_TAX)).alias("sum\_charge"))\

.orderBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS).show()

قسمتی از خروجی کوئری 1 نیز به صورت زیر می باشد :



همچنین در زیر **زمان کل کوئری های اجرا شده** در این حالت را می بینیم :

( همانطور که ذکر قبلا ذکر شد تمامی کوئری های تبدیل شده اعم از آنانکه اجرا میشوند و آنان که تبدیل شده اند ولی به دلایلی اجرا نمی شوند به طور کامل در پیوست موجود است .) :

|  |  |  |
| --- | --- | --- |
| زمان کل اجرا به دقیقه | زمان کل اجرا به ثانیه | **شماره کوئری** |
| 5 | 296 | 1 |
| 15 | 870 | 5 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

***قسمت spark + ORC + HDFS :***

در این قسمت تنها کافیست با استفاده از کل زیر که برای تک تک جداول زیر در پیوست (7) به طور کامل آورده شده اند ، از روی فایلهای پارکتی که در مرحله قبل ساختیم ، فایل ها را به فرمت ORC تبدیل کنیم . به عنوان نمونه در زیر کد این تبدیل را برای جدول Lineitem می بینیم :

from pyspark import SparkContext

from pyspark.sql import SQLContext

sc = SparkContext(appName="CSV2Parquet")

sqlContext = SQLContext(sc)

sqlContext.setConf('spark.sql.orc.impl', 'native')

lineitem = sqlContext.read.parquet("hdfs://namenode:8020/mahdi-parquet-data/lineitem.parquet")

lineitem.write.save("hdfs://namenode:8020/mahdi-orc-data/lineitem.orc", mode='overwrite', format='orc')

lineitem\_orc = sqlContext.read.orc("hdfs://namenode:8020/mahdi-orc-data/lineitem.orc")

print(lineitem\_orc.schema)

همانطور که در کد نیز مشخص است ابتدا در قسمت read محل فایل ذخیره شده پارکت را تنظیم و در قسمت اخر نیز محلی که میخواهیم فایل ORC ما در آنجا ذخیره شود را تعیین میکنیم . در انتها نیز دستور print(lineitem\_orc.schema) بدین جهت است که پس انجام کار ، شمای کلی جدول به ما نشان داده شود تا از صحت این عملیات مطمئن شویم و ضرورت خاصی برای بودن این قسمت در کدهای این تبدیل وجود ندارد. در اینجا فرمت ORC را مشخص کرده و با نوشتن mode = overwrite این امکان را میدهیم که در صورتی که قبلا همچین فایلی به طور ناقص یا اشتباه در انجا کپی شده بود ، فایل جدید روی آن ذخیره شود و نیازی به پاک کردن قبلی نباشد .

حال ادامه کافیست هر 22 تا کوئری تبدیل شده در قسمت قبلی را در این حالت نیز اجرا کنیم . تنها تفاوت ادرس محلی است که دیتافریم ها را از آنجا میخوانیم که باید به عنوان مثال از :

supplier = sqlContext.read.parquet("hdfs://namenode:8020/mahdi-parquet-data/supplier.parquet")

به:

supplier = sqlContext.read.orc("hdfs://namenode:8020/mahdi-orc-data/supplier.orc")

تغییر دهیم .

همچنین مانند زمان ساختن جداول در ORC باید یک خط کد زیر را زیر به کوئری ها اضافه کنیم :

sqlContext.setConf('spark.sql.orc.impl', 'native')

زیرا در غیر این صورت با خطایی به صورت زیر در زمان ساخت جدول یا اجرای کوئری مواجه میشویم :



به عنوان مثال کوئری شماره یک به صورت زیر در می آید :

from decimal import \*

from pyspark.sql.types import \*

from pyspark.sql import functions as F

from pyspark.sql import \*

sqlContext.setConf('spark.sql.orc.impl', 'native')

lineitem = sqlContext.read.orc("hdfs://namenode:8020/mahdi-orc-data/lineitem.orc")

func1 = lambda x , y : x \* (1-y)

func2 = lambda x ,y , z : x \* (1-y) \* (1+z)

query1 = lineitem.filter( lineitem.L\_SHIPDATE < "1998-12-01")\

.select(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS , lineitem.L\_QUANTITY , lineitem.L\_EXTENDEDPRICE , lineitem.L\_DISCOUNT , lineitem.L\_TAX )\

.groupBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS )\

.agg(F.sum(lineitem.L\_QUANTITY).alias("sum\_qty") , F.sum(lineitem.L\_EXTENDEDPRICE).alias("sum\_base\_price") , F.avg(lineitem.L\_QUANTITY).alias("avg\_qty") , F.avg(lineitem.L\_EXTENDEDPRICE).alias("avg\_price") , F.avg(lineitem.L\_DISCOUNT).alias("avg\_disc") ,

F.avg(func1(lineitem.L\_EXTENDEDPRICE , lineitem.L\_EXTENDEDPRICE )).alias("sum\_disc\_price") , F.avg(func2(lineitem.L\_EXTENDEDPRICE, lineitem.L\_DISCOUNT , lineitem.L\_TAX)).alias("sum\_charge"))\

.orderBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS).show()

همچنین در زیر **زمان کل کوئری های اجرا شده** در این حالت را می بینیم :

|  |  |  |
| --- | --- | --- |
| زمان کل اجرا به دقیقه | زمان کل اجرا به ثانیه | **شماره کوئری** |
| 4.5 | 280 | 1 |
| 13 | 790 | 5 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

***قسمت Spark + Avro + HDFS :***

در این قسمت مقداری کار با دو مرحله قبلی متفاوت و پیچیده تر است . برای تبدیل به Avro دو فایل jar را که پیوست (9) نیز آورده شده اند باید ابتدا دانلود کرده ( با دستور (wget ///////adreese site و سپس هر دو را در spark master و spark worker کپی کنیم ( با دستورcp ) . البته این نکته را نیز فراموش کردم که قبلا ذکر کنم که در تمامی مراحل قبلی و بعدی ( به جز این قسمت ) ما کاری با کانتینر spark worker نداریم و تمام کار ها و تبدیلات و اجرای کوئری خود را در محیط spark master انجام می دهیم . بعد از انجام این کار میباست از روی فایل های پارکت ساخته شده در مرحله قبلی برای هر جدول ، آن را خوانده و سپس از روی آنها به Avro تبدیل کنیم . در پیوست شماره (9) تمامی کد های مربوط به این قسمت به صورت کامل آورده شده اند . در زیر به عنوان مثال کد تبدیل nation به Avro از روی parquet را می بینیم :

from pyspark import SparkContext

from pyspark.sql import SQLContext

nation = sqlContext.read.parquet("hdfs://namenode:8020/mahdi-parquet-data/nation.parquet")

nation.write.mode('overwrite').format("com.databricks.spark.avro").save("hdfs://namenode:8020/mahdi-avro-data/nation.avro")

nation = spark.read.format("com.databricks.spark.avro").load("hdfs://namenode:8020/mahdi-avro-data/nation.avro")

همانطور که در کد بالا نیز میبینیم در ابتدا از فایل پارکت ساخته شده در مراحل قبلی عمل خواندن را انجام میدهیم و در nation میریزیم . سپس از روی nation ایجاد شده در این قسمت با دستور نوشته شده در وسط کد آن را بر ادرسی که میدهیم به صورت Avro می ریزیم .

در انتها نیز یک مرتبه فایل نوشته شده را میخوانیم تا از صحت عملیات انجام شده اطمینان یابیم . نوشتن قسمت سوم کد ضرورتی ندارد و صرفا جهت حصول اطمینان از درستی کد و عملیات صورت گرفته می باشد .

حال باید تمام 22 کوئری تبدیل شده با پایتون به اسپارک را در این قسمت نیز اجرا بگیریم تا بتوانیم خروجی ها را با هم مقایسه کنیم . تنها تفاوت نیز مانند مرحله قبلی ، تغییر آدرس خواندن دیتافریم ها می باشد که به عنوان مثال باید از :

supplier = sqlContext.read.parquet("hdfs://namenode:8020/mahdi-parquet-data/supplier.parquet")

به :

supplier = spark.read.format("com.databricks.spark.avro").load("hdfs://namenode:8020/mahdi-avro-data/supplier.avro")

تغییر پیدا کند .

به عنوان مثال کوئری کامل شماره یک در این حالت به صورت زیر می باشد :

from decimal import \*

from pyspark.sql.types import \*

from pyspark.sql import functions as F

from pyspark.sql import \*

lineitem = spark.read.format("com.databricks.spark.avro").load("hdfs://namenode:8020/mahdi-avro-data/lineitem.avro")

func1 = lambda x , y : x \* (1-y)

func2 = lambda x ,y , z : x \* (1-y) \* (1+z)

query1 = lineitem.filter( lineitem.L\_SHIPDATE < "1998-12-01")\

.select(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS , lineitem.L\_QUANTITY , lineitem.L\_EXTENDEDPRICE , lineitem.L\_DISCOUNT , lineitem.L\_TAX )\

.groupBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS )\

.agg(F.sum(lineitem.L\_QUANTITY).alias("sum\_qty") , F.sum(lineitem.L\_EXTENDEDPRICE).alias("sum\_base\_price") , F.avg(lineitem.L\_QUANTITY).alias("avg\_qty") , F.avg(lineitem.L\_EXTENDEDPRICE).alias("avg\_price") , F.avg(lineitem.L\_DISCOUNT).alias("avg\_disc") ,

F.avg(func1(lineitem.L\_EXTENDEDPRICE , lineitem.L\_EXTENDEDPRICE )).alias("sum\_disc\_price") , F.avg(func2(lineitem.L\_EXTENDEDPRICE, lineitem.L\_DISCOUNT , lineitem.L\_TAX)).alias("sum\_charge"))\

.orderBy(lineitem.L\_RETURNFLAG , lineitem.L\_LINESTATUS).show()

همچنین در زیر **زمان کل کوئری های اجرا شده** در این حالت را می بینیم :

|  |  |  |
| --- | --- | --- |
| زمان کل اجرا به دقیقه | زمان کل اجرا به ثانیه | **شماره کوئری** |
| 6.5 | 345 | 1 |
| 17 | 980 | 5 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

در زیر به مقایسه کوئری های اجرا شده در 3 حالت به کمک نمودار میپردازیم :



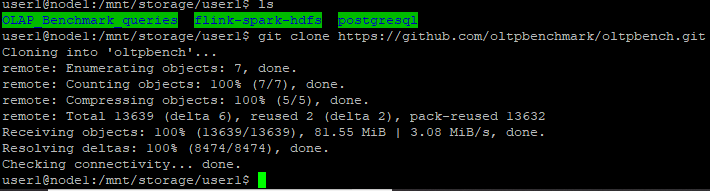
***قسمت OLTP :***

در این قسمت همانطور که میدانیم ، OLTP برای اجرای تراکنش ها ( کوئری ها با حجم کم ) مناسب است . بدین منظور ما نیاز داریم از کوئری ها و benchmark مخصوصی بهره ببریم . در ادامه مراحل نصب و راه اندازی و اجرا را ذکر خواهیم کرد .

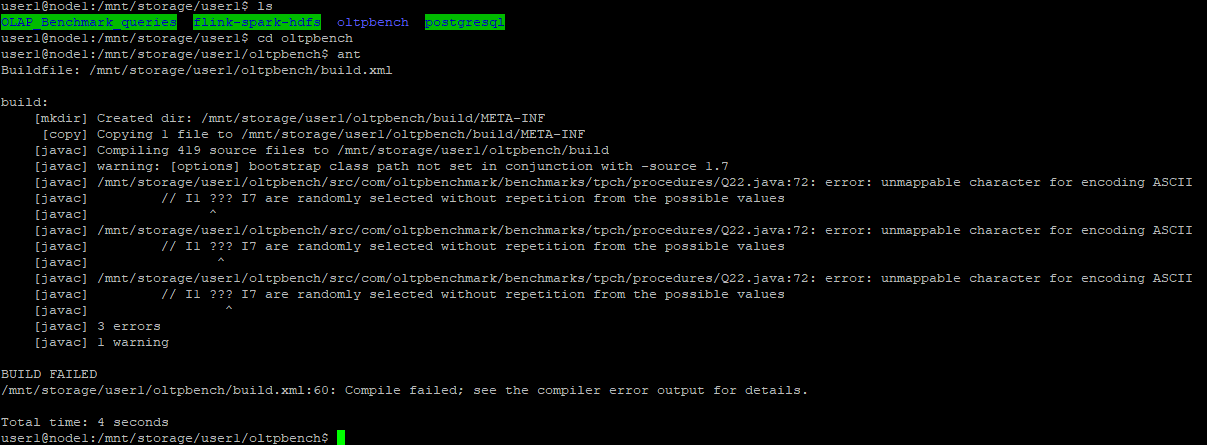
در ابتدا لازم است عمل clone را بدین صورت انجام دهیم با آدرس زیر :

git clone https://github.com/oltpbenchmark/oltpbench.git

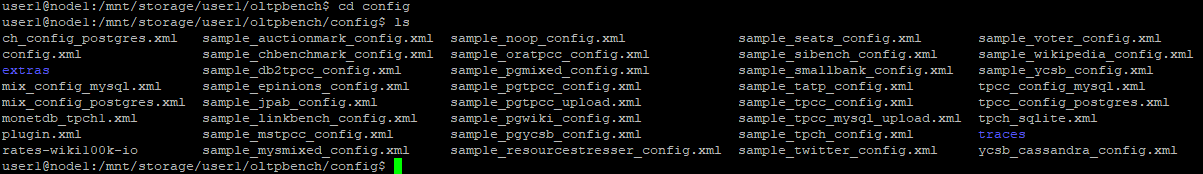
موجب دانلود oltpbench با حجم دانلود 81 مگ به صورت زیر میشود :



من در کل عملیات نصب را در آدرس /mnt/storage/user1 که متعلق به من است و اجازه نصب فایل و داده در آن را دارم انجام دادم . سپس نیاز است در صورت نصب نبودن ant آن را نصب کنیم . در مرحله بعدی نیز باید داخل دایرکتوری oltpbench ایجاد شده توسط clone شده و در آن تنها دستور ant را بزنیم . بار اول که من این عمل را انجام دادم با ارور زیر رو به رو شدم که با کمک آقای نجاتی ( مسئول سرور ) رفع شد . هر چند دلیل آن را دقیق نفهمیدیم ولی برای اجرا build.xml برای جاوا نصب شده نسخه قدیمی تر از 8 باشد ، چون در نسخه های بعدی این عمل ساپورت نمیشود و در این مورد مسئول سرور جناب آقای نجاتی به من کمک کرد که این رو متوجه بشم .



حال در مرحله بعدی باید یک فایل config را ادیت کنیم . بدین منظور وارد دایرکتوری config شده و همانطور که در شکل پایین نیز میبینیم نمونه های اولیه ای برای تنظیم config در حالات مختلف اجرا وجود دارد .



ما باید مورد sample\_tpcc\_config.xml رو ادیت کنیم .

در اجرای Transaction ها ما سه نحوه اجرا به صورت زیر داریم :

1) TRANSACTION\_SERIALIZABLE

2) TRANSACTION\_READ\_COMMITTED

3) TRANSACTION\_REPEATABLE\_READ

در مورد تفاوت این 3 مورد در درس توضیح داده شده است و از تکرار آنها در اینجا می پرهیزم .

نحوه مقدار دهی پارامتر ها و ادیت این فایل با توجه به مشخصات مد نظر ما و راهنمای موجود در قسمت 4 پروژه مشخص میشود و نتیجه همانطور که در پیوست شماره (10) نیز برای هر یک از 3 روش آمده است و در زیر نیز به عنوان مثال برای روش اول ( Transaction\_Serializable ) میبینیم ، اینگونه است :

<?xml version="1.0"?>

<parameters>

<!-- Connection details -->

<dbtype>postgres</dbtype>

<driver>org.postgresql.Driver</driver>

<DBUrl>jdbc:postgresql://0.0.0.0:32770/tpcc</DBUrl>

<DBName>tpcc</DBName>

<username>root</username>

<password>root</password>

<isolation>TRANSACTION\_SERIALIZABLE</isolation>

<!-- Scale factor is the number of warehouses in TPCC -->

<scalefactor>2</scalefactor>

<!-- The workload -->

<terminals>10</terminals>

<works>

<work>

<time>600</time>

<rate>10000</rate>

<weights>45,43,4,4,4</weights>

</work>

</works>

<!-- TPCC specific -->

<transactiontypes>

<transactiontype>

<name>NewOrder</name>

</transactiontype>

<transactiontype>

<name>Payment</name>

</transactiontype>

<transactiontype>

<name>OrderStatus</name>

</transactiontype>

<transactiontype>

<name>Delivery</name>

</transactiontype>

<transactiontype>

<name>StockLevel</name>

</transactiontype>

</transactiontypes>

</parameters>

به عنوان مثال مقدار 0.0.0.0:32770 همان شماره port داکر مربوط به پوستگره من است که اینجا آورده ام و tpcc نیز اسم دیتابیسی است که در مرحله بعدی باید در داخل داکر پوستگره ام بسازم و همچنین زمان 600 نیز زمان اجرا و امدن رکوعست هاست و عدد 10000 نیز میزان رکوعست هایی است کع ارسال میشود و می توان حتی آن را unlimited قرار داد . در اینجا ما 10 ترمینال نیز در نظر گرفتیم که می تواند مقدار دیگری داشته باشد اما بهتر است چندین ترمینال داشته باشیم .

همانطور که گفتیم پس از ادیت این فایل config با ادیتور Vim میبایستی وارد داکر پوستگره خود شده و به صورت زیر یک پایگاه داده بسازیم :

Create database tpcc ;

حال مجدد به دایرکتوری قبلی یعنی oltpbench بر میگردیم . حالا باید ابتدا بنچمارک را برای tpcc با شرایط قبلی ساخته و داده ها را درون آن لود کنیم . بدین منظور از دستور زیر استفاده میکنیم :

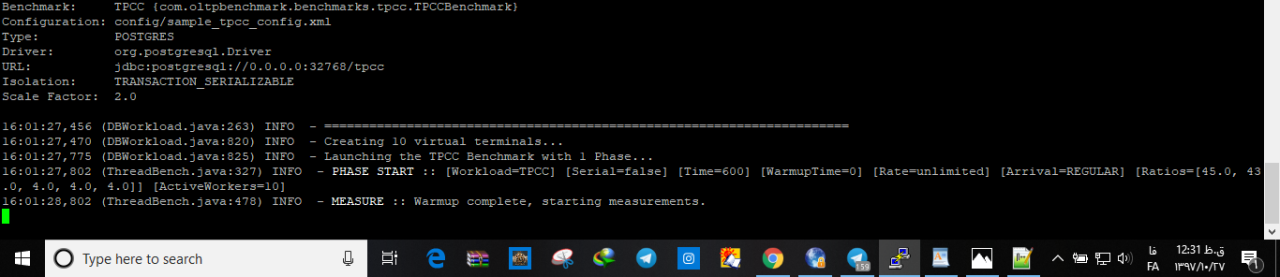
./oltpbenchmark -b tpcc -c config/sample\_tpcc\_config.xml --create=true --load=true

بعد از اینکه این عملیات به اتمام رسید ، باید به صورت زیر execute صورت بگیرد :

./oltpbenchmark -b tpcc -c config/sample\_tpcc\_config.xml --execute=true -s 5 -o outputfile

خروجی نهایی نیز در یک فایل در پوشه result به نام outputfile ذخیره میشود .

در حین اجرا با وضعیتی که در زیر میبینیم ، روبه رو میشویم :



در پایان کار خروجی را نیز میتوانیم در دایرکتوری result با نام outputfile ببینیم .

Outputfile مربوطه به هر 3 روش در پیوست به شماره (11) آورده شده است .