SQL and NoSQL Performance Comparison for big and small dataset.

**Introduction**

In the past, there was a clear difference between relational and non-relational (‘NoSQL’) database systems. However, as both types of systems have evolved, some similarities between them have appeared.

PostgreSQL, one of the most broadly used relational databases is a well-known open source project with many contributors. Besides storing relational data, it also has the ability to store semi-structured data, including XML and Json data.

MongoDB, another open source, non-relational database application stores data in flexible, JSON-like documents, meaning that fields can vary from document to document and data structure can be changed over time. Even though it has a proprietary query language, this language supports many of the same operators that SQL does (for instance, MongoDB recently added the ability to support joins).

Documentation about each system is available at the  [What is MongoDB page](https://www.mongodb.com/what-is-mongodb) and [What is PostgreSQL](https://www.postgresql.org/about/) page.

As the two types of systems have some common capabilities, a comparison between them is possible. This report describes some performance tests of traditional relational database technology versus non-relational (‘NoSQL’) database technology. I run an experiment on both PostgreSQL and MongoDB with the same data and queries, in order to compare their performance. In the following sections I describe the goals of the test, the system used, the setup (including a step by step description of the tasks, so that the experiment can be repeated), and the results obtained.

**Goal**

The goal of the test is to compare the performance of traditional relational database technology and non-relational (‘NoSQL’) database technology for storing and querying semi-structured data. In order to perform this comparison, I created a dataset in two sizes (small data set, 1GB; and large data set, 100GB) of relational data, based on the TPCH benchmark (Reference). I then stored the data in traditional relational tables (called ‘normalized data’) and in a nested, Json structure (called ‘unnormalized data’), in Postgres. I also copied the Json data to MongoDB. I then composed a series of SQL queries to be run over the relational data. The queries were then translated into SQL extended with Json functions to be run over the Json data in Postgres, and into the MongoDB query languages, based on Json, to be run over the data in MongoDB.

The test is composed of

-a run of the queries with normalized data using the small data set (1GB). Data has no indices

-a run of the queries with normalized data using the small data set (1GB). Indices are added to the data, as described below.

-a run of the queries over the unnormalized (Json) data in Postgres using the small data set (1GB). Data has no indices.

-a run of the queries over the unnormalized (Json) data in Postgres using the small data set (1GB). Indices are created on the data, as described below.

-a run of the queries of the unnormalized (Json) data in MongoDB using the small data set (1GB). Data has no indices.

-a run of the queries over the unnormalized (Json) data in MongoDB using the small data set (1GB). Indices are created on the data, as described below.

The whole experiment is then repeated with the large data set (100 GB).

Each query is run 5 times. All times were recorded, but the times reported are the average of 3 times after eliminating the slowest and fastest run.

Finally, the results were analyzed to determine the *relative* performance of each system. That is, what is of interest here is not the absolute performance but how each system did relative the the other.

**Environment:**

I ran the experiments in a server with an Intel Xeon CPU with 4 cores, each one with 12K cache, and 48 MB of RAM. The system had 2.4 TB of SCSI disk, of which 1.5 TB were assigned to the partition holding the database data. Note that one data set (‘small’) was designed to be smaller than RAM while the other (‘large’) was designed to be larger than RAM. This allowed me to compare how each system used the available RAM and the system cache, as well as how it managed dealing with data on disk.

The OS is Red Hat Enterprise Linux 6.0. I used PostgreSQL 10.4 and MongoDB 3.6.5

**Preparation**

I used the TPCH benchmark, a well-known database benchmark for Decision Support. TPCH provides a (relational) database schema and a tool to generate data for the schema in several sizes. First, I generate database with the TPCH tool, then I import these tables to PostgreSQL. After that, I generate a nested Jsonb table with Jsonb functions inside the PostgreSQL system. Finally, I export the Jsonb table to MongoDB.

**Step 1: Generate TPCH table**

1). Download TPCH package from below website to local computer.

<http://www.tpc.org/tpc_documents_current_versions/download_programs/tools-download-request.asp?bm_type=TPC-H&bm_vers=2.17.3&mode=CURRENT-ONLY>

Copy TPC\_H.zip to server with code ‘scp’.

Run below code in local terminal:

$scp /mydirectory of tpch/TPC\_H.zip server log in id

In my computer, the code is :

$ scp /Users/bonnyxin/Downloads/TPC\_H.zip [xin@dblab.cecsresearch.org:/home/xin](mailto:xin@dblab.cecsresearch.org:/home/xin)

Connect to server and unzip TPC\_H.zip:

$unzip TPC\_H.zip

2).Build dbgen tool.  
‘Readme’ in the dbgen tool has the instruction on how to build the tool.  
Make sure that PLATFORM/MACHINE is set to "LINUX" in ‘makefile.’ You can ignore the DATABASE values even if they are set.

cd TPC\_H/dbgen/

vi makefile

I changed ‘makefile’ as below:

CC = GCC

# Current values for DATABASE are: INFORMIX, DB2, TDAT (Teradata)

# SQLSERVER, SYBASE, ORACLE

# Current values for MACHINE are: ATT, DOS, HP, IBM, ICL, MVS,

# SGI, SUN, U2200, VMS, LINUX, WIN32

# Current values for WORKLOAD are: TPCH

DATABASE= SQLSERVER

MACHINE = LINUX

WORKLOAD = TPCH

#

Run ‘make ‘to generate exe file ‘dbgen’ and ‘qgen’.

make

3). Generate data.  
By default, ‘dbgen’ will create the data in ascii text files, one file for each table in TPC-H framework.  
The fields are separated by pipe "|" and file with an extension of ‘.tbl’. Run below code to generate 100GB tables.

cd TPC\_H/dbgen

./dbgen -s100 -f

4) Delete “|” to fit requirement of PostgreSQL.

PostgreSQL requires that the delimiter should not appear at the end of every line. Here is a c++ program to delete delimiter. Open it with text editor and change line about ‘lineitem.tbl’ and ‘lineitem1.tbl’ for table ‘lineitem’. Edit this c++ program 8 times for 8 tables as below then save them in local computer.

 #include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main(int argc, const char \* argv[]) {

string s;

ifstream in;

in.open("lineitem.tbl");

ofstream out;

out.open("lineitem1.tbl");

if (in.is\_open()) {

while (getline(in, s)) {

int len = s.length();

int i = len-1;

//将最后一个竖号去掉才能满足PostgreSQL的数据读取

if (s[i] == '|')

s[i] = '\n';

out << s;

}

}

out.close();

return 0;

}

Zip 8 cpp files and copy to server under TPC\_H/dbgen, where the same directory as ‘tbl’ file.

Note: to copy file from local computer to server, you need run ‘scp’ command in local terminal.

scp /Users/bonnyxin/Desktop/696/delimiter.zip xin@dblab.cecsresearch.org: /home/xin/TPC\_H/dbgen

Connect to server :

cd TPC\_H/dbgen

unzip delimiter.zip

mv /home/xin/TPC\_H/dbgen/delimiter/customer.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/part.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/orders.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/nation.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/supplier.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/lineitem.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/region.cpp /home/xin/TPC\_H/dbgen

mv /home/xin/TPC\_H/dbgen/delimiter/partsupp.cpp /home/xin/TPC\_H/dbgen

Generate ‘exe’ file:

g++ -Wall -g -o supplier supplier.cpp

g++ -Wall -g -o customer customer.cpp

g++ -Wall -g -o part part.cpp

g++ -Wall -g -o orders orders.cpp

g++ -Wall -g -o nation nation.cpp

g++ -Wall -g -o lineitem lineitem.cpp

g++ -Wall -g -o region region.cpp

g++ -Wall -g -o partsupp partsupp.cpp

Run ‘exe’ file and generate 8 tables for PostgreSQL.

./lineitem

./orders

./customer

```````

Now I got 8 new tables without delimiter ready for PostgreSQL.

orders1.tbl

customer1.tbl

lineitem1.tbl

**Step 2: generate traditional tables in PostgreSQL.**

1) Create database, under server root

cd;

createdb project2 -O xin -E UTF8 -e

2) Create table with dss.dll. This file is from TPCH package, under ‘dbgen’.

cd

psql project2;

\include /home/xin/TPC\_H/dbgen/dss.ddl



3).Copy data to PostgreSQL.

psql project2;

\copy customer from '/home/xin/TPC\_H/dbgen/customer1.tbl' WITH DELIMITER AS '|';

\copy lineitem from '/home/xin/TPC\_H/dbgen/lineitem1.tbl' WITH DELIMITER AS '|';

\copy orders from '/home/xin/TPC\_H/dbgen/orders1.tbl' WITH DELIMITER AS '|';

4). Add foreign key.

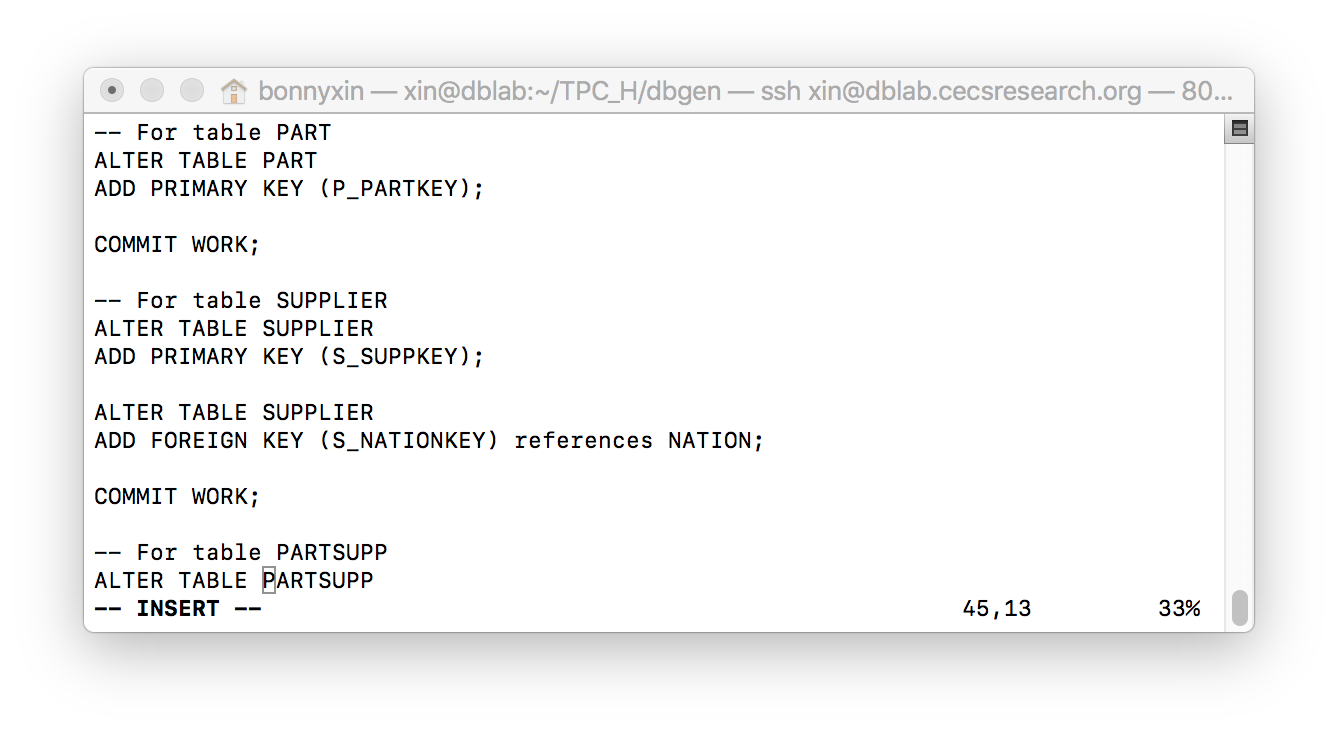
First, exit PostgreSQL, go to directory ‘dbgen’ in TPCH and modify file dss.ri with ‘vi’. Link here about [VI text editor with linux command](https://www.guru99.com/the-vi-editor.html). Below screen shot is how I modified file ‘dss.ri’.

\q

cd home/xin/TPC\_H/dbgen

vi dss.ri

i



esc

shift+ZZ

Next, run ‘dss.ri’ in PostgreSQL.

cd

psql project2;

\include /home/xin/TPC\_H/dbgen/dss.ri;

/describe tables

\d+ lineitem

Now we have 8 traditional tables in database named ‘project2’ in PostgreSQL ready for the test. These tables in PostgreSQL are: part, supplier, partsupp, customer, nation ,lineitem, region and orders. In this experiment, we just work with table ‘customer’, ‘orders’ and ‘lineitem’. Therefore, when I copy data to PostgreSQL, I just copied above 3 tables to save disk space.

**Step 3: combine tables ‘customer’, ‘orders’, ‘lineitem’ to one Jsonb table.**

First, I created a Jsonb table for ‘orders’ and ‘lineitem’, then embed ‘lineitem’ to ‘orders’.

create table order\_lineitem( data jsonb );

insert into order\_lineitem select row\_to\_json(t)

from (

select orders.\*,json\_agg(row\_to\_json(lineitem)) as lineitem

from orders

left join lineitem on o\_orderkey=l\_orderkey

group by o\_orderkey

) t;

Second, with the same way to embed the generated Jsonb table ‘order\_lineitem’ to ‘customer’.

create table customer\_or\_lm( value Jsonb );

insert into customer\_or\_lm select row\_to\_json(t)

from (

select customer.\*, json\_agg(row\_to\_json(order\_lineitem)) as orders

from customer

left join order\_lineitem on to\_Jsonb(c\_custkey)=data->'o\_custkey'

group by c\_custkey

) t;

Now the Jsonb table ‘customer\_or\_lineitem’ is ready in PostgreSQL for test. It takes 24 hours to generate this big size table.

**Step 4 Transfer Jsonb table to MongoDB.**

First, export Jsonb table from PostgreSQL, the code below is to copy *data only* from PostgreSQL as Json format.

psql project2

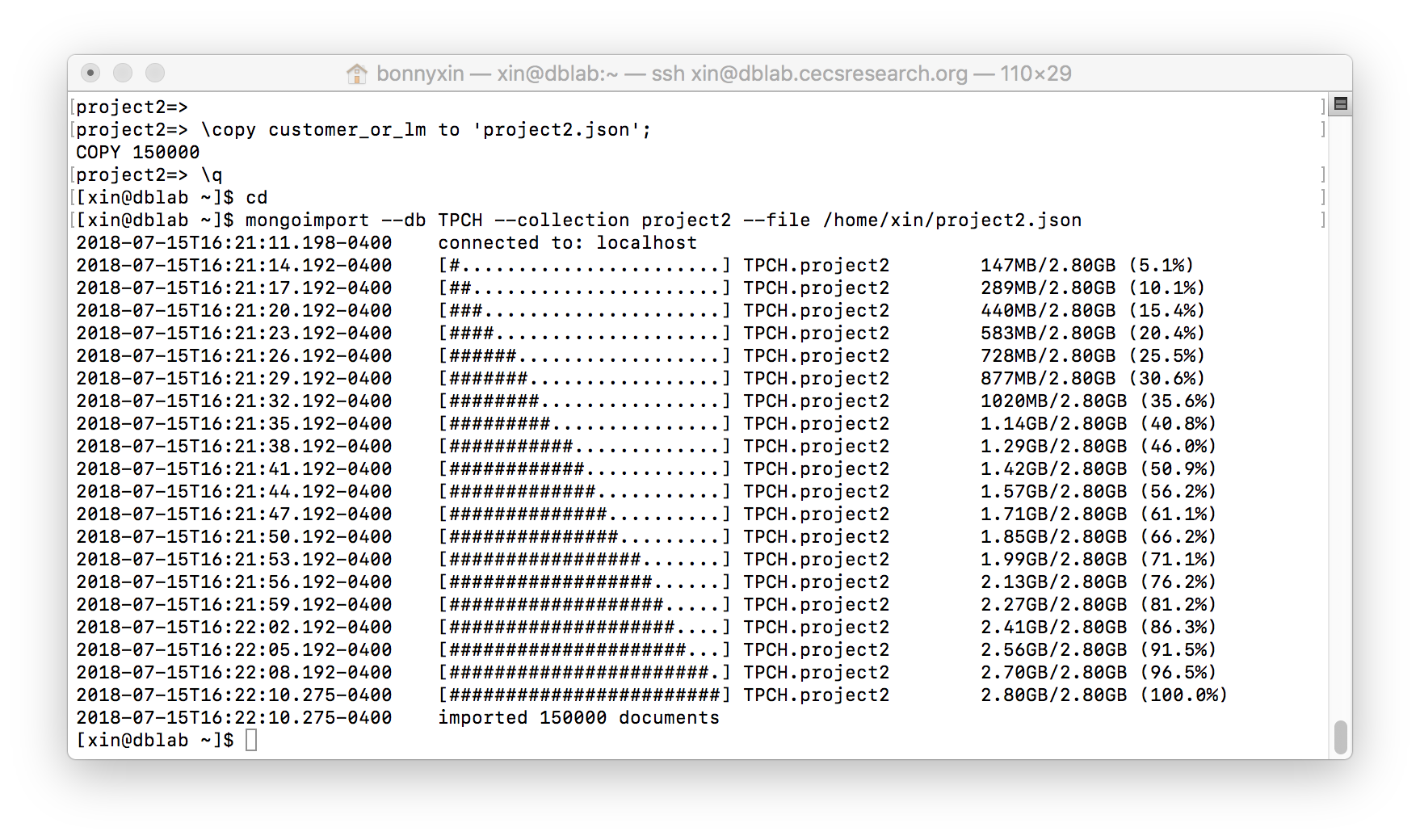
\copy customer\_or\_lm to 'project2.json';

\q

Second, import ‘project2.json’ to MongoDB.

cd;

mongoimport --db TPCH --collection project2 --file /home/xin/project2.json



In the end, turn on MongoDB, you will find the generated table.

mongo

use TPCH

show collections

**Implementation**

**1). About the queries.**

I used a list of queries derived from those of the TPCH benchmark. Some queries were taken from the benchmark as they are, while others were created from scratch. There are three versions of each query in the list. One version is written in SQL, over the traditional database in PostgreSQL; another one is written in SQL with Json functions, over the Json data in Postgres; the last one is written in MongoDB's language, over the data in MongoDB.

In each version of query list, there are 8 queries, named query1, 2,3,4,5,6,7,8. Each query has 3 extension names. ‘-a’ returns a large result, ‘-b’ returns a medium result, ‘-c’ returns a small output. The size of the result is controlled by modifying constants in comparison operations so as to restrict the data that qualifies for an answer. For instance, in 1GB dataset, output of ‘1-a’ is 29754 rows, output of ‘1-b’ is 5294 rows, output of ‘1-c’ is 136 rows.

Some queries go 'deep' into the structure (in this case, this means asking for details on ‘lineitem’, which is the innermost embedded item), such as query 7. Some queries require joins (in SQL), such as query 3,4,6,8. Some queries are to compare the running time with or without aggregation, such as query 1 and 2, query 3 and 4.

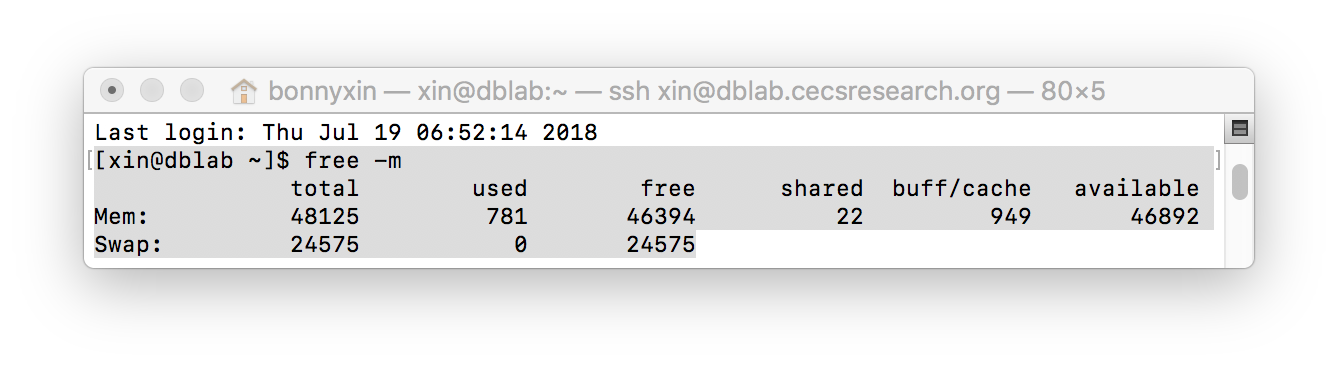
You can find the query list in appendix.

**2). Change parameters of PostgreSQL.**

First, we need change some parameters in PostgreSQL to make sure we can use most RAM. You can find details about [How to tune your PostgreSQL Server](https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server).

To find out available RAM in my server.

$ free -m



Since the available RAM is 46892MB. I will set my ‘effective\_cache\_size’ to 25GB, and ‘work\_mem’ to 50MB in PostgreSQL.

$ psql

ALTER SYSTEM SET effective\_cache\_size = '25GB';

ALTER SYSTEM SET work\_mem = '50MB';

After changing above parameters, stop PostgreSQL and restart it. We can find the parameters have been changed:

\q

sudo su;

systemctl stop postgresql-10.service;

sync;

echo 3 > /proc/sys/vm/drop\_caches;

systemctl start postgresql-10.service;

exit;

psql

show effective\_cache\_size;

show work\_mem;

**3.) Run test without index.**

This is the list of tasks:

for each query size (small, medium, large):

    for each regular SQL query q on that size:

         Run q in Postgres 5 consecutive times and record the times

         Run it with EXPLAIN once more and record the query plan.

         stop Postgres, flush cache, restart Postgres.

Once this is done, repeat the whole thing with Json SQL queries.

Once this is done, repeat the whole thing in MongoDB.

Task explanation:

**3.1).** In PostgreSQL, for each regular SQL query, run it 5 consecutive times and record the times, command ‘\timing on’ will show the execution time.

Then add ‘EXPLAIN ANALYSE’ in front of query to get the plan. Here is an example of query 1-a:

EXPLAIN ANALYSE

select c\_name, c\_address, c\_phone

from customer

where c\_acctbal < 10000

and c\_mktsegment = 'AUTOMOBILE';

After recording the times and plan, let us flush cache with below code:

First, quit PostgreSQL and turn to root,

\q

sudo su;

Second, stop PostgreSQL.

systemctl stop postgresql-10.service;

Third, flush the file system buffer:

sync;

Fourth, clear PageCache, dentries and inodes:

echo 3 > /proc/sys/vm/drop\_caches;

Fifth, start PostgreSQL and exit root.

systemctl start postgresql-10.service;

exit;

In the end turn to project2 to of PostgreSQL.

psql project2;

\o out1.txt;

Command ‘\o out1.txt’ is to export output to a txt file named out1.txt. That will save displaying time, due to the large outputs of some queries.

Appendix ‘queries\_psql.sql’ is the script with queries and about how to clear cache.

3.2). For Json table in PostgreSQL, we run queries and flush cache with same procedure as above.

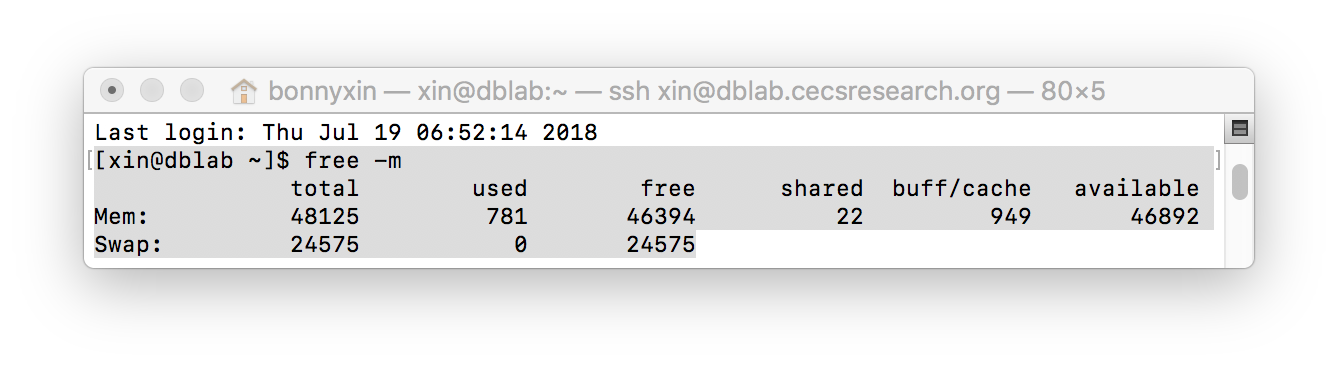
You can find details from appendix ‘queries\_jsonb.sql’.

3.3).For MongoDB.

Make sure the RAM is mostly available before we start a test.

cd

free -m



If the available RAM is less than ¾ of total RAM, clear cache.

3.3.1). First, run the test.

Turn on MongoDB. Set up the profiler level to 2. Then the profiler collects data for all operations.

mongo ;

use TPCH;

db.setProfilingLevel(2);

Run the query once.

db.getCollection("project2").find(

{

"c\_mktsegment" : "AUTOMOBILE",

"c\_acctbal" : {

"$lt" : NumberLong(1000)

}

},

{

"c\_name" : "$c\_name",

"c\_address" : "$c\_address",

"c\_phone" : "$c\_phone"

}

);

Then find out the running time.

db.system.profile.find().sort({$natural:-1}).limit(1).pretty();

Above code shows the last operation. Then we can find the running time:

"millis" : 120.

Run above code 5 times then clear the cache.

3.3.2).Second, clear the cache.

We need to empty the system cache and restart the MongoDB server for each query too. Doing this is similar to PostgreSQL.

Quit MongoDB:

exit;

Go back to root

sudo su;

Stop MongoDB:

systemctl stop mongod.service;

Flush the file system buffer:

sync;

Clear PageCache, dentries and inodes:

echo 3 > /proc/sys/vm/drop\_caches;

Restart MongoDB:

systemctl start mongod.service;

exit;

mongo;

use TPCH;

Appendix ‘query\_MongoDB.sql’ is the code to run test, make sure to repeat each query and get time for 5 consecutive times then clear the cache.

**4). Test with indexes.**

This means creating an index on each attribute that has a selection on it, both in PostgreSQL and in MongoDB. Also, in PostgreSQL relational tables, we can cluster the ‘order’ table by ‘customer ‘key, and the ‘lineitem’ table by ‘order key’, to see if that improves join performance. We can then rerun all the queries and compare to the previous times.

4.1). For traditional dataset.

4.1.1). Create cluster indexes:

This is done in 2 stages: create an index first, and then cluster the table using the index.

CREATE INDEX orders\_id\_index ON orders (o\_custkey);

CREATE INDEX lineitem\_id\_index ON lineitem (l\_orderkey);

CLUSTER orders USING orders\_id\_index;

CLUSTER lineitem USING lineitem\_id\_index;

Since this physically reorders the data in the table, to make sure the optimize is aware of the changes, we need run ANALYZE as below:

analyze orders;

analyze lineitem;

4.1.2) Run test and get the record.

4.2) For Jsonb tables in PostgreSQL.

4.2.1). Create indexes first:

CREATE INDEX index\_cktsegment ON customer\_or\_lm ((value->>'c\_mktsegment'));

CREATE INDEX index\_cacctbals ON customer\_or\_lm((value->>'c\_acctbal'));

CREATE INDEX index\_cname ON customer\_or\_lm((value->>'c\_name'));

CREATE INDEX index\_oclerk ON customer\_or\_lm((value-'orders'->'data'->>'o\_clerk'));

CREATE INDEX index\_ototalprice ON customer\_or\_lm((value-'orders'->'data'->>'o\_totalprice'));

CREATE INDEX index\_oorderpriority ON customer\_or\_lm((value-'orders'->'data'->>'o\_orderpriority'));

CREATE INDEX index\_cnationkey ON customer\_or\_lm((value->>'c\_nationkey'));

CREATE INDEX index\_oorderdate ON customer\_or\_lm((value-'orders'->'data'->>'o\_orderdate'));

CREATE INDEX index\_lcommitdate ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_commitdate'));

CREATE INDEX index\_lreceiptdate ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_receiptdate'));

CREATE INDEX index\_lreturnflag ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_returnflag'));

CREATE INDEX index\_llinestatus ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_linestatus'));

CREATE INDEX index\_lshipdate ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_shipdate'));

CREATE INDEX index\_ltax ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_tax'));

CREATE INDEX index\_ldiscount ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_discount'));

CREATE INDEX index\_lshipmode ON customer\_or\_lm((value-'orders'->'data'->'lineitem'->>'l\_shipmode'));

4.2.2).Run test and record the times.

4.3).For MongoDB collections.

4.3.1). Create indexes as below:

db.project2.createIndex({"c\_mktsegment":1})

db.project2.createIndex({"c\_acctbal":1})

db.project2.createIndex({"c\_name":1})

db.project2.createIndex({"orders.data.o\_clerk":1})

db.project2.createIndex({"c\_nationkey":1})

db.project2.createIndex({"orders.data.o\_totalprice":1})

db.project2.createIndex({"orders.data.o\_orderpriority":1})

db.project2.createIndex({"orders.data.o\_orderdate":1})

db.project2.createIndex({"orders.data.lineitem.l\_commitdate":1})

db.project2.createIndex({"orders.data.lineitem.l\_receiptdate":1})

db.project2.createIndex({"orders.data.lineitem.l\_returnflag":1})

db.project2.createIndex({"orders.data.lineitem.l\_linestatus":1})

db.project2.createIndex({"orders.data.lineitem.l\_shipdate":1})

db.project2.createIndex({"orders.data.lineitem.l\_tax":1})

db.project2.createIndex({"orders.data.lineitem.l\_discount":1})

db.project2.createIndex({"orders.data.lineitem.l\_shipmode":1})

4.3.2). Run tests and record the results.

**Analysis of Results**

I average the time records of a, b and c for each query of 1GB dataset and get data sheet and graph as below.

**TABLE 1**



Graph 1

In graph1, ‘SQL and NoSQL Performance Comparison for 1GB dataset’, axe is the running time, milliseconds. From the graph we can see for most queries, the orange line is below the others. As the graph measures time, a lower line indicates less time, that is, faster performance. So ‘psql with cluster indexes’, the orange line in the graph, runs faster than the other systems in the experiment. According to the experimental results, ‘psql no index’ beats ‘psql with cluster indexes’ on query 5 and 6. Since they are both queries in the traditional dataset on PostgreSQL, we can conclude that, for small dataset, such as 1GB tables, with big RAM, 50GB, traditional normalize dataset is the best option.

TABLE 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Comparison for psql with cluster index and other SQL and NoSQL | | | | | | |
| **Query No.** | **psql  no index** | **psql with cluster indexes** | **jsonb without index** | **jsonb with index** | **mongodb without index** | **mongodb with index** |
| **Query 1** | 1.02824412 | 1 | 146.763877 | 88.8509148 | 1.3610328 | 0.02872892 |
| **Query 2** | 1.04442791 | 1 | 121.284236 | 63.4849874 | 5.80645771 | 2.29484322 |
| **Query 3** | 4.63610979 | 1 | 168.21539 | 165.074563 | 11.5786632 | 10.1297791 |
| **Query 4** | 2.97540729 | 1 | 94.1501921 | 95.2623606 | 18.2193896 | 15.8048939 |
| **Query 5** | 0.46635911 | 1 | 5.45239661 | 5.40010231 | 27.9288379 | 28.8427897 |
| **Query 6** | 0.83827853 | 1 | 9.52481084 | 2.86129493 | 13.6801165 | 13.7396739 |
| **Query 7** | 1.01033438 | 1 | 9.80850166 | 9.85282309 | 26.2490054 | 25.9451389 |
| **Query 8** | 1.02763838 | 1 | 10.6638407 | 10.7120155 | 50.6389903 | 50.7714087 |

Graph 2

We also compare running time of ‘psql with cluster indexes’ with others. We can see that the difference of running time between ‘psql no index’ and ‘psql with cluster indexes’ is not significant from TABLE 2. Comparing two unnormalized datasets, we see that for queries 1 to 4, which run on small tables in the traditional dataset, (‘orders’ and ‘customers’), running time of Jsonb is 63 to 168 times of ‘psql with cluster indexes’, and MongoDB is 0.028 to 18 times of ‘psql with cluster indexes’. For, queries 5 to 8, which run on big tables (‘lineitem’), running time of SQL(Json) is 2 to 8 times of running time of traditional dataset, and running time of NoSQL(MongoDB) is 13 to 50 times of traditional dataset. Thus we get the conclusion that NoSQL(MongoDB) beats SQL(Json) on simple queries, but SQL(Json) beats NoSQL on complex queries.

For 100GB dataset, I compared below performance as below: normalize dataset without index in PostgreSQL; normalize dataset with indexes in PostgreSQL; unnormalized (Json) data without index in PostgreSQL; unnormalized (Json) data with indexes in PostgreSQL; unnormalized (Json) data without index in MongoDB.

**TABLE 3**



TABLE 3 is the average running time for 100GB dataset. Graph 3 is generated from data in TABLE 3. The Unit is millisecond.

According to Graph 3, it is hard to say with tool is best for all queries.

**Graph 3**

For simple query without ‘group by’, such as query 1 and 3, MongoDB run fast. The difference is significate. Running time of ‘psql no index’ and ‘psql with cluster indexes’ is 35 to 53 times of the running time of ‘MongoDB without index’; the running time of ‘jsonb no index’ and ‘jsonb with index’ is 1,872 to 16,851 times of the running time of ‘Mongodb without index’. Please see details in TABLE 4.

TABLE 4



For simple query with ‘group by’, traditional normalize dataset runs fast. According to TABLE 5, there is no significate difference between ‘psql no index’ and ‘psql with cluster indexes’. However, traditional dataset in PostgreSQL runs tens to hundreds time faster than unnormalize dataset in query 2 and 4.

TABLE 5



For complex queries, such as query 5 to 8. Running time of MongoDB is 2 to 5 times of running time of ‘Jsonb with indexes’. Thus, NoSQL is not a good option for complex query in big dataset.

There is no significate difference between the running time of ‘jsonb with index’ and ‘Jsonb without index’. We can see from TABLE 6, running time of query 5, a query with correlated subquery, are interesting. Running time of ‘psql with cluster indexes’ is 14 times of the running of ‘jsonb with indexes’. For query 6,7,8 running time of ‘psql no index’ beats other systems.

According to the data, we can make a conclusion for big dataset, MongoDB (NoSQL) runs fast on simple query without ‘group by’, traditional dataset is a good option on complex query without correlated subquery.

TABLE 6



Graph 4

**Conclusion**

In this test, I generated a 1GB dataset and a 100GB dataset with TPCH tool. Load two datasets to PostgreSQL as both traditional normalize tables and unnormalize tables (Jsonb). And import the unnormalized dataset (Json) to MongoDB. I run 4 simple queries and 4 complex queries to compare the performance of three types of technology.

According to the analyze of result, I will choose unnormalize dataset (Json) in PostgreSQL with indexes when I work with complex queries with correlated subquery in a big dataset; choose MongoDB when I work with simple queries without ‘group by’ in both big and small dataset; choose traditional normalize dataset (PostgreSQL) for small dataset, as well as for queries without correlated subquery in big dataset.

**Further research**

These data are very valuable, the test on big dataset is time consuming. We can go deeper for these data in the further, there is a lot data surprised me, such as why NoSQL takes so long on big dataset, what is the solution for that? We can also compare the performance of MongoDB with indexes, or with GIN indexes in the future.

**Appendix A**

Appendix A is the scripts of code about how to run the test in ‘code’ folder.

1. ‘dss.ddl’ is for create traditional tables in Postgresql.
2. ‘dss.ri’ is to add foreign keys for traditional tables in Postgresql.
3. ‘query\_mongodb.sql’ is for MongoDB.
4. ‘queries\_psql.sql’ works for traditional dataset in PostgreSQL.
5. ‘queries\_jsonb.sql’ works over Jsonb table in PostgreSQL.
6. ‘indext\_creation.txt’ .
7. ‘drop\_index.txt’ .

**Appendix B**

Appeindix B is a zip folder named ‘delimiter. Zip’ to get rid of all delimiters for 8 ‘.tbl’ table generated by TPCH tool.

**Appendix C**

Appendix C is all time records and plans for 1GB dataset.

1. timing\_record\_1GB.xlsx
2. plan\_ json\_w/indexes\_1GB.pdf
3. plan\_json\_noindex\_1GB.pdf
4. plan\_mongodb\_w/indexes\_1GB
5. plan\_mongodb\_without\_index\_1GB
6. plan\_psql\_no\_ indexes\_1GB.pdf
7. plan\_psql\_w/cluster\_indexes\_1GB.pdf

**Appendix D**

Appendix D is all time rcords and plans for 100 GB datset.

1. timing\_record\_100GB.xlsx
2. plan\_ json\_w/indexes\_100GB.pdf
3. plan\_json\_noindex\_100GB.pdf
4. plan\_mongodb\_w/indexes\_100GB
5. plan\_mongodb\_noindex\_100GB
6. plan\_psql\_no\_ indexes\_100GB.pdf
7. plan\_psql\_w/cluster\_indexes\_100GB.pdf