COMPARING MONGODB, NEO4j AND ORACLE 11g

OBJECTIVES

This is the main lab on NOSQL. As we have discussed in this course, each NOSQL solution appeared to cover a certain set of requirements. Thus, in a real-world scenario, you first need to gather your system requirements and then analyse what database management system can help you better.

As Yogi Berra said, *"In theory there's no difference between theory and practice. But in practice, there is."* Thus, we aim at grasping all these differences by means of a threefold (hands-on) comparison. In this lab you will learn the strong and weak points of a document-store (MongoDB) and a graph database (Neo4j) with regard to a relational database (Oracle 11g). We will measure these pros and cons as follows:

- Design: How easy is to design a database with these systems,
- Tuning: How easy is to tune up the server,
- Performance: Under what circumstances each of them achieve a better performance,
- Other non-quantitative (but qualitative) measures such as ease to deploy (time invested to make it run), ease to develop code on top of it, friendliness, linkage with high-level languages, and any other aspect you may consider relevant.

REQUIRED KNOWLEDGE

This laboratory session subsumes the entire course but, specially, sessions on key-value and document-stores. Also, the MongoDB seminar will be of great help.
You can choose your lab mate for this lab. Just be sure to let the lecture know with the team creation 4 event.

TOOLS

This lab requires MongoDB, Neo4j and Oracle 11g.

TRAINING (ACTIVITIES TO DO DURING THE WEEK)

There is no training for this lab.

DELIVERABLES

This session is equivalent to three sessions (i.e., subsumes the document stores lab, the graph databases lab and the comparison lab). However, we will let you choose the amount of workload you feel more comfortable with. Accordingly, each group must choose if:

- They will perform the comparison between Oracle 11g and MongoDB (see the exercise statement for further details). Maximum mark: 7 out of 10.
- Perform the comparison between Oracle 11g and Neo4j (see the exercise statement for further details). Maximum mark: 8 out of 10.
- Perform the threefold comparison: Oracle 11g, Neo4j and MongoDB (see the exercise statement for further details). Maximum mark: 10 out of 10.

There will be a lab delivery and also a checkpoint (to see your progress). The dates are as follows:

- 22nd April, 2013. There is no lab session but you are already supposed to be working on this exercise during that week.
- 29th April, 2013. There is no lab session but you are already supposed to be working on this exercise during that week.
- 6th May, 2013. First checkpoint. In slots of 10 minutes you will be presenting your findings up to this date. You are supposed to show the following:
    - For the first system chosen (MongoDB or Neo4j) the database design (i.e., structure of the documents or graph),
    - Show to the lecturer you have already set MongoDB or Neo4j up and it is running in your laptop.
    - Run a wee script to insert data and run a couple of queries on top of it.

  A few days before this session lab a document will be published showing what is exactly expected from you at this point.

- Final delivery on a day (to be agreed) in June (further details to be announced).

EXERCISE STATEMENT

We will consider the TPC-H benchmark (http://www.tpc.org/tpch/) for this exercise.

In appendix A you will find the database schema and table descriptions to be used in this exercise. In appendix B you will find some rules to populate the database (these rules MUST be enforced at any moment) and a set of queries. This is our reference database setting that we will use to compare the three solutions.

Now, you are asked to proceed as follows for each solution.

Oracle 11g

Part 1: The normalized schema

- Create the "normalized" version of this schema in Oracle 11g (you can create the tables using SQL Developer). A SQL script can be found in the additional material of this session (TPC-H.txt).
- Create a java application accessing the Oracle 11g database you just created using JDBC. The JDBC driver for Oracle 11g can be downloaded here.
- From the JDBC application, insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**). Measure the time (i.e., store the time before and after the insertion script).
- From your JDBC application run every SQL query in appendix B. Run each of them 5 times (to avoid collisions with other groups firing queries at the same time) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples more (**do not drop the previous inserted tuples and meet the insertion rules in Appendix B**) and measure the insertion time again.
- From your JDBC application run, again, every SQL query in appendix B. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the normalized Oracle template (see appendix C).

Note: Insertions must be done through JDBC (to simulate data-shipping). Otherwise, if you run this script from SQL Developer you would be simulating query-shipping and Oracle 11g would be in advantage with regard to MongoDB and Neo4j.

Note2: To avoid the network latency, you must run the definitive test over Oracle 11g from any of the UPC lab rooms (otherwise, you will be incurring in a huge network communication overhead from your home).

Part 2: The normalized schema

Now, clean the database (dump all data) and from SQL Developer tune the database schema as much as you can (check the amount of time you should devote to this lab and

proceed accordingly[1]). **Any tuning** is valid (i.e., partitioning, indexes, clustering, denormalization, etc.). Your aim is to speed up all queries as much as possible.

The only two rules are: you cannot change the database setting between queries and the Oracle 11g account space limit (40Mb).

Therefore, if you choose some tuning, you must run all queries with the same tuning and, once you have inserted data, you will have around 30% of the total space for tuning issues.

Now:

- Modify your JDBC application to adapt it to your new database schema and insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) from scratch. Measure the insertion time.
- From your JDBC application run every SQL query in appendix B (adapted to your new schema). Run each of them 5 times (to avoid collisions with other groups firing queries at the same time) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) more and measure the insertion time again.
- From your JDBC application run, again, every SQL query. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the tuned Oracle template (see appendix C).

MongoDB

Now, let us see how easy is to design a similar scenario with MongoDB and how easy is to improve the database performance with this document-oriented database.

First, set MongoDB up in your laptop (you will have to run this exercise during the final delivery in front of the lecture, so you better do it on a laptop!). There are plenty of manuals with lots of information about this, for example, this is the most basic one: http://docs.mongodb.org/manual/installation/

Also, check this one, from SQL to MongoDB, a nice discussion:
http://docs.mongodb.org/manual/reference/sql-comparison/

And this one about design considerations:
http://docs.mongodb.org/manual/core/data-modeling/

Anyway, this is up to you; decide which manuals suit better your needs. Maybe some other manuals help you better. You are expected to need a couple of hours to set the environment and get familiar with the basics.

[1] This lab should imply around 20-25h of work time so please, do not overdo (unless you really want to).

Then, proceed as follows:

Part 1: The normalized schema

- Simulate a "normalized" version of the TPC-H schema in MongoDB (i.e., each table is a collection and each table row is a document in that collection).
- Create an application accessing the MongoDB database you just created using a high level language driver (e.g., Java). You are free to use the driver and high level language you prefer. Java is suggested but again, this is up to you (use one you feel comfortable with). The drivers can be found here.
- From your application, insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) from scratch. Measure the insertion time.
- From your application run every SQL query in appendix B (adapted to your MongoDB database). Run each of them 5 times (to avoid collisions with other processes running at the same time) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples more (**remember to meet the insertion rules in Appendix B**) and measure the insertion time again.
- From your application run, again, every query. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the normalized MongoDB template (see appendix C).

Part 2: The denormalized schema

Part 1 will help you to get familiar with MongoDB and its API. Now, it is time for tuning. **Any tuning** is valid (i.e., tuning Mongo server parameters –bear in mind te MongoDB seminar-, indexes, denormalization, etc.). Your aim is to speed up all queries as much as possible.

The only rule is: you cannot change the database setting between queries (the amount of space used here is up to you). Therefore, if you choose some tuning, you must run all queries with the same tuning.

Once you have normalized your database, proceed as always. Thus:

- From your application, insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) from scratch. Measure the insertion time.
- From your application run every SQL query in appendix B (adapted to your MongoDB database). Run each of them 5 times (to avoid little variations due to the OS) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples more (**remember to meet the insertion rules in Appendix B**) and measure the insertion time again.
- From your application run, again, every query. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the tuned MongoDB template (see appendix C).

Neo4j

Now, let us see how easy is to design a similar scenario with Neo4j and how easy is to improve the database performance with this graph database.

First, set Neo4j up in your laptop (you will have to run this exercise during the final delivery in front of the lecture, so you better do it on a laptop!). There are plenty of manuals with lots of information about this, for example, this is the most basic one: http://www.neo4j.org/download

Also, check this nice tutorial for Neo4j embedded in Java applications:
http://docs.neo4j.org/chunked/stable/tutorials.html

You can decide to use Cypher (the declarative language created with Neo4j to query graphs). Here is a manual on how to use Cypher from java:
http://docs.neo4j.org/chunked/stable/tutorials-cypher-java.html

Some data modeling examples are presented here:
http://docs.neo4j.org/chunked/stable/data-modeling-examples.html

Finally, you can get a free book on graph databases (and rather interesting) at:
http://www.neo4j.org/learn

Anyway, this is up to you; decide which manuals suit better your needs. You are expected to need a couple of hours to set the environment and get familiar with the basics.
Then, proceed as follows:

Part 1: The normalized schema

- Simulate a "normalized" version of the TPC-H schema in Neo4j (i.e., each table is a node and FK-PK relationships are graph edges). You can use the shell to do this step.
- Create an application accessing the Neo4j database you just created using a high level language driver (e.g., Java). You are free to use the driver and high level language you prefer. Java is suggested but again, this is up to you (use one you feel comfortable with). The drivers can be found here.
- From your application, insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) from scratch. Measure the insertion time.
- From your application run every SQL query in appendix B (adapted to your Neo4j database). Run each of them 5 times (to avoid collisions with other processes running at the same time) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples more (**remember to meet the insertion rules in Appendix B**) and measure the insertion time again.
- From your application run, again, every query. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the normalized Neo4j template (see appendix C).

Part 2: The denormalized schema

Part 1 will help you to get familiar with Neo4j and its API. Now, it is time for tuning. **Any tuning** is valid (i.e., tuning Neo4j server parameters, indexes, denormalization, etc.). Your aim is to speed up all queries as much as possible.

The only rule is: you cannot change the database setting between queries (the amount of space to be used is up to you). Therefore, if you choose some tuning, you must run all queries with the same tuning.

Once you have normalized your database, proceed as always. Thus:

- From your application, insert 20.000 lineitem tuples (**remember to meet the insertion rules in Appendix B**) from scratch. Measure the time before and after.
- From your application run every SQL query in appendix B (adapted to your Neo4j database). Run each of them 5 times (to avoid little variations due to the OS) and choose the minimum time (and write it down for each query).
- Insert 20.000 lineitem tuples more (**remember to meet the insertion rules in Appendix B**) and measure the insertion time again.
- From your application run, again, every query. Run each of them 5 times and choose the minimum time (and write it down).
- Fill the tuned Neo4j template (see appendix C).

APPENDIX A

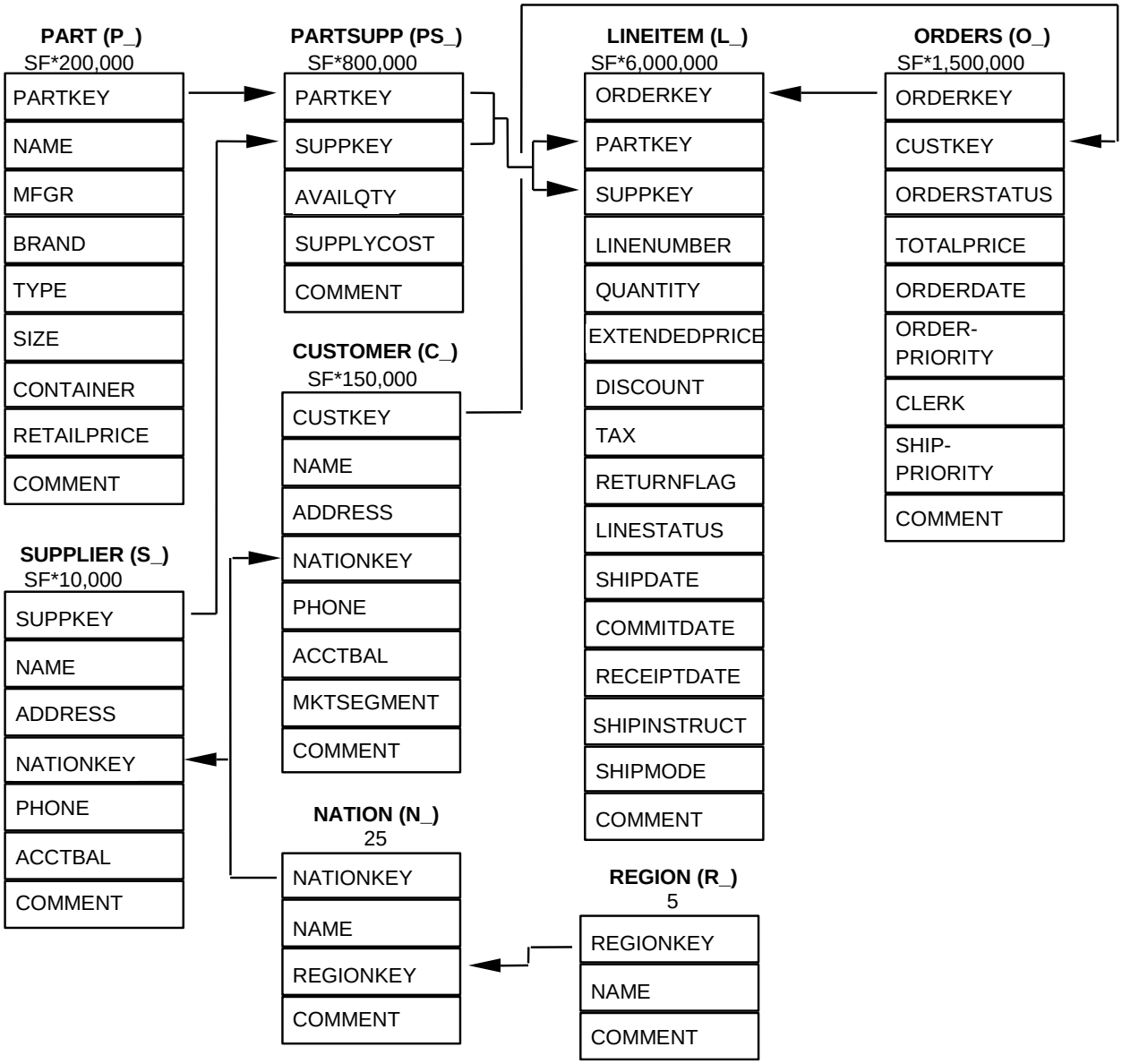
In this appendix you will find a graphical representation of the TPC-H schema and further explanations on the schema you may want to check. In the additional material of this session you will find a SQL script to create such schema as it is shown here.

IMPORTANT NOTE: This appendix introduces you to the TPC-H benchmark. Do not follow the insertion rules stated here (just follow those in Appendix B). This is just for your information, to know more about the schema, nothing else.

1.2 Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 2: The TPC-H Schema.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

1.3 Datatype Definitions

1.3.1 The following datatype definitions apply to the list of columns of each table:

- **Identifier** means that the column must be able to hold any key value generated for that column and be able to support at least 2,147,483,647 unique values;

Comment: A common implementation of this datatype will be an integer. However, for SF greater than 300 some column values will exceed the range of integer values supported by a 4-byte integer. A test sponsor may use some other datatype such as 8-byte integer, decimal or character string to implement the identifier datatype;

- **Integer** means that the column must be able to exactly represent integer values (i.e., values in increments of 1) in the range of at least -2,147,483,646 to 2,147,483,647.
- **Decimal** means that the column must be able to represent values in the range -9,999,999,999.99 to +9,999,999,999.99 in increments of 0.01; the values can be either represented exactly or interpreted to be in this range;
- **Big Decimal** is of the Decimal datatype as defined above, with the additional property that it must be large enough to represent the aggregated values stored in temporary tables created within query variants;
- **Fixed text, size N** means that the column must be able to hold any string of characters of a fixed length of N.

Comment: If the string it holds is shorter than N characters, then trailing spaces must be stored in the database or the database must automatically pad with spaces upon retrieval such that a CHAR_LENGTH() function will return N.

- **Variable text, size N** means that the column must be able to hold any string of characters of a variable length with a maximum length of N. Columns defined as "variable text, size N" may optionally be implemented as "fixed text, size N";
- **Date** is a value whose external representation can be expressed as YYYY-MM-DD, where all characters are numeric. A date must be able to express any day within at least 14 consecutive years. There is no requirement specific to the internal representation of a date.

Comment: The implementation datatype chosen by the test sponsor for a particular datatype definition must be applied consistently to all the instances of that datatype definition in the schema, except for identifier columns, whose datatype may be selected to satisfy database scaling requirements.

1.3.2 The symbol SF is used in this document to represent the scale factor for the database (see Clause 4:).

1.4 Table Layouts

1.4.1 Required Tables

The following list defines the required structure (list of columns) of each table.

The annotations ‘Primary Key’ and ‘Foreign Key’, as used in this Clause, are for information only and do not imply additional requirements to implement **primary key** and **foreign key** constraints (see Clause 1.4.2).

PART Table Layout

Column Name	Datatype Requirements	Comment
P_PARTKEY	identifier	SF*200,000 are populated
P_NAME	variable text, size 55	
P_MFGR	fixed text, size 25	

P_BRAND	fixed text, size 10
P_TYPE	variable text, size 25
P_SIZE	integer
P_CONTAINER	fixed text, size 10
P_RETAILPRICE	decimal
P_COMMENT	variable text, size 23

Primary Key**:** P_PARTKEY

SUPPLIER Table Layout

Column Name	Datatype Requirements	Comment
S_SUPPKEY	identifier	SF*10,000 are populated
S_NAME	fixed text, size 25	
S_ADDRESS	variable text, size 40	
S_NATIONKEY	Identifier	Foreign Key to N_NATIONKEY
S_PHONE	fixed text, size 15	
S_ACCTBAL	decimal	
S_COMMENT	variable text, size 101	

Primary Key**:** S_SUPPKEY

PARTSUPP Table Layout

Column Name	Datatype Requirements	Comment
PS_PARTKEY	Identifier	Foreign Key to P_PARTKEY
PS_SUPPKEY	Identifier	Foreign Key to S_SUPPKEY
PS_AVAILQTY	integer	
PS_SUPPLYCOST	Decimal	
PS_COMMENT	variable text, size 199	

Primary Key**:** PS_PARTKEY, PS_SUPPKEY

CUSTOMER Table Layout

Column Name	Datatype Requirements	Comment
C_CUSTKEY	Identifier	SF*150,000 are populated

C_NAME	variable text, size 25	
C_ADDRESS	variable text, size 40	
C_NATIONKEY	Identifier	Foreign Key to N_NATIONKEY
C_PHONE	fixed text, size 15	
C_ACCTBAL	Decimal	
C_MKTSEGMENT	fixed text, size 10	
C_COMMENT	variable text, size 117	

Primary Key**:** C_CUSTKEY

ORDERS Table Layout

Column Name	Datatype Requirements	Comment
O_ORDERKEY	Identifier	SF*1,500,000 are sparsely populated
O_CUSTKEY	Identifier	Foreign Key to C_CUSTKEY
O_ORDERSTATUS	fixed text, size 1	
O_TOTALPRICE	Decimal	
O_ORDERDATE	Date	
O_ORDERPRIORITY	fixed text, size 15	
O_CLERK	fixed text, size 15	
O_SHIPPRIORITY	Integer	
O_COMMENT	variable text, size 79	

Primary Key**:** O_ORDERKEY

Comment: Orders are not present for all customers. In fact, one-third of the customers do not have any order in the database. The orders are assigned at random to two-thirds of the customers (see Clause 4:). The purpose of this is to exercise the capabilities of the DBMS to handle "dead data" when joining two or more tables.

LINEITEM Table Layout

Column Name	Datatype Requirements	Comment
L_ORDERKEY	identifier	Foreign Key to O_ORDERKEY
L_PARTKEY	identifier	Foreign key to P_PARTKEY, first part of the compound Foreign Key to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY	Identifier	Foreign key to S_SUPPKEY, second part of the compound Foreign Key to (PS_PARTKEY,

		PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER	integer	
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_TAX	decimal	
L_RETURNFLAG	fixed text, size 1	
L_LINESTATUS	fixed text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	fixed text, size 25	
L_SHIPMODE	fixed text, size 10	
L_COMMENT	variable text size 44	

Primary Key**:** L_ORDERKEY, L_LINENUMBER

NATION Table Layout

Column Name	Datatype Requirements	Comment
N_NATIONKEY	identifier	25 nations are populated
N_NAME	fixed text, size 25	
N_REGIONKEY	identifier	Foreign Key to R_REGIONKEY
N_COMMENT	variable text, size 152	

Primary Key**:** N_NATIONKEY

REGION Table Layout

Column Name	Datatype Requirements	Comment
R_REGIONKEY	identifier	5 regions are populated
R_NAME	fixed text, size 25	
R_COMMENT	variable text, size 152	

Primary Key**:** R_REGIONKEY

APPENDIX B

Find here the queries and insert patterns composing the database workload. This workload is expressed in SQL. Thus, you will have to adapt it to MongoDB and Neo4j (also to your tuned version in Oracle 11g).

General rules about queries

Your queries SHOULD NOT retrieve the empty answer. All of them must retrieve tuples (otherwise, it might be immediate for Oracle to realize about this checking the catalog).

Query 1:

```
SELECT 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

FROM lineitem

WHERE l_shipdate <= 'date'

GROUP BY l_returnflag, l_linestatus

ORDER BY l_returnflag, l_linestatus;
```

Constraint: 'date' must be an existing date in the database.

Query 2:

```
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone,
s_comment

FROM part, supplier, partsupp, nation, region

WHERE p_partkey = ps_partkey AND s_suppkey = ps_suppkey AND p_size = [SIZE]
AND p_type like '%[TYPE]' AND s_nationkey = n_nationkey AND n_regionkey =
r_regionkey AND r_name = '[REGION]' AND ps_supplycost = (SELECT
min(ps_supplycost) FROM partsupp, supplier, nation, region WHERE p_partkey =
ps_partkey AND s_suppkey = ps_suppkey AND s_nationkey = n_nationkey AND
n_regionkey = r_regionkey AND r_name = '[REGION]')

ORDER BY s_acctbal desc, n_name, s_name, p_partkey;
```

Constraint: [size], [type] and [region] must be existing values in the database.

Query 3:

```
SELECT l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue,
o_orderdate, o_shippriority

FROM customer, orders, lineitem

WHERE c_mktsegment = '[SEGMENT]' AND c_custkey = o_custkey AND l_orderkey =
o_orderkey AND o_orderdate < '[DATE]' AND l_shipdate > '[DATE]'

GROUP BY l_orderkey, o_orderdate, o_shippriority

ORDER BY revenue desc, o_orderdate;
```

Constraint: [segment] and [date] and both dates must be existing values in the database.

Query 4:

```
SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue

FROM customer, orders, lineitem, supplier, nation, region

WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND l_suppkey =
s_suppkey AND c_nationkey = s_nationkey AND s_nationkey = n_nationkey AND
n_regionkey = r_regionkey AND r_name = '[REGION]' AND o_orderdate >= date
'[DATE]' AND o_orderdate < date '[DATE]' + interval '1' year

GROUP BY n_name

ORDER BY revenue desc;
```

Constraint: [date] and [region] must be existing values in the database.

Insertion rules:

When inserting data, you must meet the following constraints (failing to follow them will be considered cheating, since the performance can be drastically affected by the kind of data inserted).

In the previous appendix you have seen a graphical representation of the TPC-H schema. Below each table name, you will see something of the kind: `SF*NUMBER`. Each insertion batch you must insert 20.000 `lineitems` (i.e., `SF = 0'00333333`). You must enforce the proportion of `lineitem` with regard to the other tables. For example, you must insert 0'00333333 * 1.500.000 = 5.000 `orders`, and so on. If decimal numbers appear (e.g., for `partsupp` it will be 0'00333333 * 800.000 = 2666'6), **round that number down (i.e., 2666)**.

You can generate random values but enforcing the following constraints according to the kind of datatype:

- `Integer`: All of them must have, at least, 4 digits.
- `Varchar2(x)`: It must contain exactly x/2 characters.
- `Number(x,y)`: It must contain, at least, x/2 digits.
- `Date`: A random date (the Oracle format is '12/12/01' by default).

Furthermore:

- For each attribute, at most, only a 10% of NULL values can be generated.
- FKs and PKs must be enforced at data level, whenever possible, but you can decide not to do it at schema level. In other words, the FK-PK must be preserved for the generated data but you are not forced to implement it in the schema.
- The data distribution you did for Oracle must be preserved for MongoDB and Neo4j (for the sake of equality).

APPENDIX C

These are the templates you will be asked to deliver in the final delivery. Besides these templates, note you are asked to deliver the following for each database setting:

- A graphical representation of the schema,
- The queries adapted to such schema,
- The script used to run the tests demanded in the exercise statement.

Furthermore, for the final delivery you will need to fill a survey about your main findings (further details and a template will be published a few days before the final delivery).

TEMPLATE FOR ORACLE 11g (Normalized Template)

Keep track of the following information when working with the normalized version of Oracle 11g:

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

TEMPLATE FOR ORACLE 11g (Tuned Template)

Keep track of the following information when working with the tuned version of Oracle 11g:

List, **in a comprehensive an d precise manner**, all the tuning you carried out. In other words, any change you made to the initial CREATE TABLEs script (**attach to this document a graphical representation of the tuned schema, the SQL queries adapted to it and the script used to populate and query the database**):

Now, keep track of the same data as before:

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

TEMPLATE FOR MONGODB (Normalized Template)

Keep track of the following information when working with the *normalized* version of MongoDB:

List here any relevant comment you would like to make (**attach to this document a graphical representation of your denormalized schema, the queries adapted to it and the script used to populate and query the database**):

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

TEMPLATE FOR MONGODB (Tuned Template)

Keep track of the following information when working with the tuned version of MongoDB:

List, **in a comprehensive and precise manner**, all the tuning you carried out. In other words, any change you made to the initial *normalized* schema (**attach to this document a graphical representation of your denormalized schema, the queries adapted to it and the script used to populate and query the database**):

Now, keep track of the same data as before:

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

TEMPLATE FOR NEO4j (Normalized Template)

Keep track of the following information when working with the *normalized* version of Neo4j:

List here any relevant comment you would like to make (**attach to this document a graphical representation of your *denormalized* schema, the queries adapted to it and the script used to populate and query the database**):

Now, keep track of the same data as before:

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

TEMPLATE FOR NEO4j (Tuned Template)

Keep track of the following information when working with the tuned version of Neo4j:

List, **in a comprehensive an d precise manner**, all the tuning you carried out. In other words, a precise description of your occurrence schema (**attach to this document a graphical data representation, the queries adapted to it and the script used to populate and query the database**):

Now, keep track of the same data as before:

- Amount of time needed for inserting the first batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____

- Amount of time needed for inserting the second batch of data: _____

Minimum time to execute Q1: _____

Minimum time to execute Q2: _____

Minimum time to execute Q3: _____

Minimum time to execute Q4: _____

Average time to execute the 4 queries: _____