

ADM PROJECT 3

RUNNING THE PROJECT

To initialize the CSV data and the database, run the script `init.sh` .

1.1 THE WORKLOAD:

We invented 7 queries. We tried to select different entities, different level of complexities and different types of queries and results (Aggregates, Listings, ...). We tried to invent queries that may simulate real-life needs for company reports.

Assumptions:

- The `fromcurrencycode` can be different than USD, even if the current data only USD appears.
- Using the currency code to perform the query is ok.
- We assume that the field "saleslastyear" of the "salespersons" table is always in the same currency, hence summable.
- We use `businessentityid` as the main field to identify a sales person because we have no personal information. In the ideal case we should use the personal data as well.
- We assume that using the "productid" is good enough to describe a product.

q1: a report of the fluctuation of a currency (XXX) rate against the dollar in a specified time period.

```
SELECT cr.fromcurrencycode,
       cr.tocurrencycode,
       cr.averagerate
FROM   sales.currencyrate cr
WHERE  cr.fromcurrencycode = 'USD'
AND    cr.tocurrencycode = 'XXX'
AND    cr.currencyratedate BETWEEN 'YYY' AND 'ZZZ'
```

q2: a report of the average rate of a currency (XXX) rate against the dollar in a specified time period.

```
SELECT cr.fromcurrencycode,
       cr.tocurrencycode,
       AVG(cr.averagerate)
FROM   sales.currencyrate cr
WHERE  cr.fromcurrencycode = 'USD'
AND    cr.tocurrencycode = 'XXX'
AND    cr.currencyratedate BETWEEN 'YYY' AND 'ZZZ'
GROUP BY cr.fromcurrencycode, cr.tocurrencycode
```

q3: calculates the total amount of sales made last year by all the sales people.

```
SELECT  SUM(saleslastyear)
FROM    sales.salesperson
```

q4: gives as result the number of customers that a sales person has come in contact through his career.

```
SELECT  sp.businessentityid,
COUNT(distinct customerid) as customersnum
FROM    sales.salesperson sp
LEFT JOIN sales.salesterritoryhistory sth
ON sp.businessentityid = sth.businessentityid
LEFT JOIN sales.salesterritory st
ON sth.territoryid = st.territoryid
LEFT JOIN sales.customer c
ON c.territoryid = st.territoryid
GROUP BY sp.businessentityid
```

q5: describes for every territory the products that had been sold and in which quantity.

```
SELECT  st.name, sod.productid, SUM(sod.orderqty)
FROM    sales.salesterritory st
JOIN    sales.salesorderheader soh
ON st.territoryid = soh.territoryid
JOIN    sales.salesorderdetail sod
ON soh.salesorderid = sod.salesorderid
GROUP BY st.territoryid, productid
```

q6: a report of the orders id for every territory.

```
SELECT  st.name, soh.salesorderid
FROM    sales.salesterritory st
JOIN    sales.salesorderheader soh
ON st.territoryid = soh.territoryid
```

q7: is an aggregation of the sales for every territory.

```
SELECT  st.name, COUNT(soh.salesorderid)
FROM    sales.salesterritory st
LEFT JOIN sales.salesorderheader soh
ON st.territoryid = soh.territoryid
WHERE st.name = 'United Kingdom'
GROUP BY st.name
```

1.2: LOGICAL DESIGN

For the schema, refer to the file schema.sql.

We tried to design the Cassandra logical schema to satisfy the requirements of the queries with the best efficiency. Sometimes we loaded more data than necessary to allow more flexibility and more detailed queries.

We need four tables:

currencies to answer q1 and q2.

```
CREATE TABLE currencies (  
    fromcurrencyname text,  
    tocurrencyname text,  
    date timestamp,  
    averagerate float,  
    PRIMARY KEY ((fromcurrencyname, tocurrencyname), date)  
);
```

"currencies" is a dynamic column family. The Partition key is made of the two columns "fromcurrencyname" and "tocurrencyname". The Clustering key is the date of the rate (column "date").

salespersons to answer q3 and q4.

```
CREATE TABLE salespersons (  
    id int PRIMARY KEY,  
    saleslastyear float,  
    customerscount int  
);
```

"salespersons" is a skinny column family. The Primary key is "id".

territorysales to answer q5.

```
CREATE TABLE territorysales(  
    territoryname text,  
    productid int,  
    quantity int,  
    PRIMARY KEY(territoryname, productid)  
);
```

"territorysales" is a dynamic column family. The Partition key is "territoryname". The Clustering key is "productid".

territoryorders to answer q6 and q7.

```
CREATE TABLE territoryorders(  
    territoryname text,  
    salesorderid int,  
    PRIMARY KEY(territoryname, salesorderid)  
);
```

"territoryorders" is a dynamic column family. The Partition key is "territoryname". The Clustering key is "salesorderid".

1.3: IMPLEMENTATION

We designed the column families to represent the logical schema.

We developed 4 python scripts (create_currencies.py, create_salespersons.py, create_territoryorders.py, create_territorysales.py) to convert the data from the CSV source to the target CSV. Efficiency was not a main concern here. In a real world scenario our approach would have been to use a SQL (with the queries presented in paragraph 1.1) to extract the data. We prepared to CQL to present the correct results with the best possible efficiency.

CQL QUERIES

The CQL queries (with non-significant parameters) would be:

q1:

```
SELECT fromcurrencyname, tocurrencyname, date, averagerate  
FROM currencies  
WHERE fromcurrencyname = 'USD'  
AND tocurrencyname = 'EUR'  
AND date >= '2014-04-01'  
AND date < '2014-05-01';
```

q2:

```
SELECT fromcurrencyname, tocurrencyname, AVG(averagerate)  
FROM currencies  
WHERE fromcurrencyname = 'USD'  
AND tocurrencyname = 'EUR'  
AND date >= '2014-04-01'  
AND date < '2014-05-01';
```

q3:

```
SELECT SUM(saleslastyear)  
FROM salespersons;
```

This requires an access to multiple partitions. Performances may be suboptimal.

q4:

```
SELECT id, customerscount  
FROM salespersons;
```

q5:

```
SELECT territoryname, productid, quantity  
FROM territoriesales;
```

q6:

```
SELECT territoryname, salesorderid  
FROM territoryorders;
```

q7:

```
SELECT territoryname, COUNT(salesorderid)  
FROM territoryorders  
WHERE territoryname = 'United Kingdom';
```

1.4: PHYSICAL ORGANIZATION

We used the commands: "nodetool cfstats sales" as a shell command to get information about the memory and disk used to store the column families. "nodetool cfstats flush" as a shell command to persist the keyspace from memory to the disk. "tracing on" as a cqlsh command to get information about the queries run.

By accessing the column families before the "flush" command is run, the queries access only the MEMTABLEs. If we persist the column families with the "flush" command, the SSTABLEs are accessed.

Only 1 SSTABLE is created for every column family.

When the data are flush and persisted the results we see for the queries are:

- q1, q2: Only 1 sstable and 1 partition is accessed. 30 live cells are found. The result is loaded in memory.
- q3, q4: Only 1 sstable and 1 partition is accessed. On sequential scan is performed. 17 live cells are found. The result is loaded in memory.
- q5: Only 1 sstable is accessed. When the query is executed in the cqlsh interface, the data are paginated. For every new page ("More...") a new session is created.

- q6: Only 1 sstable and 1 partition is accessed. On sequential scan is performed. The result is loaded in memory.
- q7: The partial result is kept in memory. The process keeps on reading new live cells (100 every time) and updates the result in memory.

If the data are not flushed, the execution follows a similar path but the data are read from the memtable.

No tombstones cells are read in all cases because no data has been ever been deleted in this namespace.

TIME ESTIMATE:

- 30min: Understanding the project requirements
- 2h: Familiarization with Cassandra, CQL and documentation reading
- 30min: Familiarization with the DB schema and CSV data
- 1h: Invention of the workload, preparation and testing (via postgres) of the queries.
- 2h 30min: Preparation of the Cassandra schema and scripts to convert and load the data
- 30min: Preparation and testing of the CQL queries
- 2h 30min: Documentation
- 1h 30min: Polishing and wrapping up of the project files

ADDITIONAL INFO

[Github URL](#)