Exam 2020

June 16, 2023

Exercise 0.1. Imagine you want to design a system to maintain data about citizens and doubt whether to use HBase or MongoDB. Precisely, for each citizen, it will store: their personal data (with pID), their city data (with cID) and their employment data. We also know the workload (i.e., queries and frequency of execution) is as follows:

- Q_1**:** average salary in Barcelona (50% frequency) - information obtained from the set of city and employment data.
- Q_2: average weight for you people (less than 18 years old) (45% frequency) - information obtained from the set of personal data.
- Q_3: number of VIPs (5% frequency) - information obtained from the set of personal data.

Discuss your choice of technology and data model, without pre-computing the results of the queries. Clearly specify the structure of the data (i.e., tables/collections, keys, values, etc.), trade-offs, and assumptions made.

- joining, which can be less efficient.
- Q2: average weight for young people (less than 18 years old) (45% frequency): This query requires a secondary index on age to perform efficiently. MongoDB, with its built-in support for secondary indexes, may have an advantage. However, HBase can also support secondary indexes with some additional setup and design considerations.
- Q3: number of VIPs (5% frequency): This query requires searching personal data based on a specific attribute, which is doable efficiently in both MongoDB and HBase.

Given the workload and the need for complex queries, MongoDB might be a slightly better choice due to its rich querying capabilities and built-in support for secondary indexes. However, if the volume of data is enormous and you need extreme scalability, HBase might be a better choice due to its distributed nature and high write throughput.

- Data model for MongoDB:
  - Personal data collection: { "_id" : pID, "name": "", "age": , "weight": , "VIP_status": }
  - City data collection: { "_id" : cID, "city_name": "", "citizen_ids": [list of pID] }
  - Employment data collection: { "_id": uniqueID, "pID": , "salary": , "job_title": }
- Data model for HBase:
  - Personal data table: Row key: pID, Column Family: Personal_Info: {"name", "age", "weight", "VIP_status"}
  - City data table: Row key: cID, Column Family: City_Info: {"city_name", "citizen_ids"}
  - Employment data table: Row key: uniqueID, Column Family: Employment_Info: {"pID", "salary", "job_title"}

Exercise 0.2. In relational algebra, the antijoin operator ($\triangleright$) is defined as the complement of the semijoin on the primary keys (PKs). Formally, assuming A and B are the PLs of R and S, respectively, then

$$R \triangleright S = R \setminus R \ltimes_{A=B} S$$

Provide the MapReduce pseudo-code implementation of the antijoin operator. Assume the existence of the operator $\oplus$ to concatenate strings, $prj_{att}(s)$ to project attribute att from the tuple s, and $input(s)$ to decide the origin (i.e., R or S) from a tuple s.

$$R \triangleright S \implies \begin{cases} map\,(key\ k, value\ v) \mapsto & \\ [(v\,(A), k \oplus v \oplus R)] & if\ input\,(k \oplus v) = R \\ [(v\,(B), k \oplus v \oplus S)] & if\ input\,(k \oplus v) = S \\ reduce\,(key\ ik, vset\ ivs) \mapsto & \\ ivs\,(v) & if\ \ [ivs\,(-1)]\,.unique == R \\ \emptyset & else \end{cases}$$

Exercise 0.3. Consider three files containing the following kinds of data:

```
Employees.txt
EMP1,CARME,400000,MATARO,DEPT1,PROJ1
EMP2,EULALIA,150000,BARCELONA,DEPT2,PROJ1
EMP3,MIQUEL,125000,BADALONA,DEPT1,PROJ3

Projects.txt
PROJ1,IBDTEL,TV,1000000
PROJ2,IBDVID,VIDEO,500000
PROJ3,IBDTEF,TELEPHONE,200000
PROJ4,IBDCOM,COMMUNICATIONS,2000000

Departments.txt
DEPT1,MANAGEMENT,10,PAU CLARIS,BARCELONA
DEPT2,MANAGEMENT,8,RIOS ROSAS,MADRID
DEPT4,MARKETING,3,RIOS ROSAS,MADRID
```

Provide the ordered list of Spark operations (no need to follow the exact syntax, but just the kind of operation and main parameters) you would need to obtain the departments with at least one employee which have all of their employees assigned to the same project. The result must include department number. Save the results in "output.txt". In the previous example, the result should be "DEPT2".

```
from pyspark.sql.functions import count

emp = spark.read.format("csv").load("employees.txt", header=False, inferSchema='true', sep
    =',')
proj = spark.read.format("csv").load("projects.txt", header=False, inferSchema='true', sep
    =',')
dept = spark.read.format("csv").load("departments.txt", header=False, inferSchema='true', sep
    =',')

emp = emp.toDF("eID","eName","eSal","eCity","eDept","eProj")
proj = proj.toDF("pID","pName","pType","pBud")
dept = dept.toDF("dID","dType","dNumber", "whatisthat", "dCity")

emp_dept = emp.join(dept, emp.eDept == dept.dID, 'inner')
emp_dept_proj = emp_dept.join(proj, emp_dept.eProj == proj.pID, 'inner')

dept_proj = emp_dept_proj.select('dID', 'pID').distinct()

result = dept_proj.groupBy('dID').agg(count('pID').alias("nProjs"))
result = result.filter(result.nProjs == 1).select('dID')

result.write.save(path='./output.txt', format='csv', header='false')
```

Exercise 0.4. Assume we ingest a stream with an event every time a ticket is sold at a theater. Precisely, the stream has the structure (movieID, theaterID, timestamp, price). Next, we ingest the following ordered set of events:

- Event 1: (m3, t4, 12h, 10$)
- Event 2: (m1, t2, 13h, 17$)
- Event 3: (m2, t4, 14h, 11$)
- Event 4: (m4, t1, 15h, 8$)
- Event 5: (m1, t3, 16h, 9$)
- Event 6: (m3, t4, 17h, 5$)
- Event 7: (m6, t1, 18h, 15$)
- Event 8: (m5, t2, 19h, 12$)
- Event 9: (m7, t5, 20h, 17$)
- Event 10: (m1, t1, 21h, 11$)

Which theaters would be considered heavy hitters (using the approximate method) considering a required frequency of 33%? Provide a detailed answer (i.e., describe the process).

Since we look for 33% appearance, we take a 3 spots structure. The steps are:

ID	Count	ID	Count	ID	Count	ID	Count	ID	Count
t4	1	t4	1	t4	2	t4	2	t4	1
		t2	1	t2	1	t2	1	t3	1
						t1	1		

ID	Count	ID	Count	ID	Count	ID	Count	ID	Count
t4	2	t4	2	t4	1	t4	1	t1	1
t3	1	t3	1	t2	1	t2	1		
		t1	1			t5	1		

Therefore, the only theater considered a heavy hitter is t1. Note that t1 only appears 3 times, i.e., 30%, but it is consider a heavy hitter anyway. Note also that there is no real heavy hitter with 33% threshold.

Exercise 0.5. What problem do Data Lakes solve?

The problem of having a single input/truth, which arises in traditional architectures in which the data model is fixed beforehand for the whole organization.