Exam 2019

June 17, 2023

Exercise 0.1. In the framework of the RUM conjecture, name six data structures we saw in the course (not concrete tool implementations) and place them in the corresponding category:

- Read optimized: BTree, Hash Index
- Write optimized: LSMTree, Hash Map
- Memory optimized: Bitmap Index, run-length encoded columns

Exercise 0.2. Which is the main problem in having replicas, and which is the innovation introduced by some NOSQL tools to solve it.

- Problem: the problem is maintaining consistency between different replicas, i.e., reflecting updates in the system correctly in all replicas. All this without hindering the performance of the system (for example if we need to block the whole system for every update, the cost would be very high, and scalability would decrease).
- Solution: eventual consistency is presented by NOSQL system as the solution to this problem, by enabling updates to be transmitted at different paces through the system, asynchronously. This is usually achieved by voting mechanisms.

Exercise 0.3. Suppose you have a hash function whose range has size 100 (i.e., D=100), and a Consistent Hash structure with 5 machines (M1..5) whose identifiers map to values h(M1)=0, h(M2)=20, h(M3)=40, h(M4)=60, h(M5)=80. What happens if you have an object mapped to value h(O)=90?

A consistent hash structure is circular, and each object is stored in the next machine in this circular structure. Since 90>80, it would go to the next machine in the circule, i.e., the first machine, M1.

Exercise 0.4. Suppose you implement a system to store images in hundreds of machines with thousands of users using HBase with a single column-family. These images taken at time VT belong to a person P who tags each with a single subject S (e.g., family, friends, etc.) and are concurrently uploaded into the system at time TT in personal batches containing multiple pictures of different subjects taken at different times. Each person can then retrieve all his/her pictures of one single subject that were taken after a given time. Precisely define the key you would use if you exclusively prioritize (i.e., do not consider any other criteria) …

- Load balancing on ingestion: VT,S,P
  - Assumptions: VT is going to be different for every picture, S will coincide for some picture, P and TT will coincide for all pictures in each, but we need P to retrieve the user fast.
- Load balancing on querying: VT,S,P
  - Assumptions: again, we want the leftmost part of the key to be as unique as possible, this is the best option.
- I/O on ingestion: P,S,VT
  - Assumptions: grouping by users would make ingestion very localized. S cannot go first because it is likely that many many users would use the same subject a lot.
- I/O on querying: P,S,VT

– the user's pictures and subjects would be very close to one another.

Exercise 0.5. Consider two files containing the following kinds of data:

```
Employees.txt
EMP1;RICARDO;250000€;MADRID;SITE2
EMP2;EULALIA;150000€;BARCELONA;SITE1
EMP3;MIQUEL;125000€;BADALONA;SITE3
EMP4;MARIA;175000€;MADRID;SITE2
EMP5;ESTEBAN;150000€;MADRID;SITE4

Departments.txt
SITE1;DPT.MANAGEMENT;FLOOR10;ST.PAU CLARIS;BARCELONA
SITE2;DPT.MANAGEMENT;FLOOR8;ST.RIOS ROSAS;MADRID
SITE3;DPT.MARKETING;FLOOR1;ST.PAU CLARIS;BARCELONA
SITE4;DPT.MARKETING;FLOOR1;ST.RIOS ROSAS;MADRID
SITE5;DPT.MARKETING;FLOOR5;ST.MARTI PUJOL;BADALONA
```

Provide the ordered list of Spark operations (no need to follow the exact syntax, but just the kind of operation and main parameters) you'd need to retrieve the list of department IDs for those departments with sites in all cities where employees live (these employees can be even from other departments). Save the results in "output.txt". In the previous example, the result should be "DPT.MARKETING", because it has sites in all the three cities where there are employees (i.e., MADRID, BARCELONA and BADALONA). However, "DPT.MANAGEMENT" should not be in the result, because it does not have any site in BADALONA, where EMP3 lives.

1. emp = spark.read.csv('Employees.txt').delimiter=';',header='false').toDF().columns(['eID',"eName','eSalary','eCity','eS
2. dept = spark.read.csv(Departments.txt').delimiter=';',header='false').toDF().columns(['dSite','dName','dFloor','dStre
3. emp_cities = emp.select('eCity').distinct().groupBy().agg(collect_list('dCity')).as('eCities')
4. dept_cities = dept.select('dName','dCity').groupBy('dName').agg(collect_list('dCity')).as('dCities')
5. result = dept_cities.filter(size(array_except(emp_cities['eCities'],dept_cities['dCities']))==0).select('dName')
6. result.write.save('./output.txt', format='csv', header='false')

Another solution:

1. emp = spark.read.csv('Employees.txt').delimiter=';',header='false').toDF().columns(['eID',"eName','eSalary','eCity','eS
2. dept = spark.read.csv(Departments.txt').delimiter=';',header='false').toDF().columns(['dSite','dName','dFloor','dStre
3. dept__cities = dept.join(emp, dept.dCity = emp.eCity, 'inner').select('dName', 'eCity').distinct().groupBy('dName').a
   #Note that these cities are shared with employees! This is important because else this approach would not work
4. eCities_count = emp.select('eCity').distinct().count()
5. result = dept.filter(dept.nCities == eCities_count).select('dName')
6. result.write.save('./output.txt',format='csv',header='false')