Exam 2022

June 15, 2023

Exercise 0.1. Compare a B-tree and an LSM-tree in the context of the RUM conjecture (i.e., as an answer to this question, three brief explanations of the form "From the perspective of X, Y-tree is better than Z-tree, because of this and that." are expected).

R: B-Tree is better than the LSM for Reads, because of the ordered structure that allows to fetch the correct leaves very efficiently, as well as scanning them in order.

U: The structure of the LSM tree makes it very suitable for writes, since the writes are done in-memory until a size threshold is surpassed.

M: in terms of memory, LSM tree usually performs better, too, because it can leverage compression techniques on older data, while focusing on the most recent and probably accessed data.

Exercise 0.2. Given a file with 3.2GB of raw data stored in an HDFS cluster of 50 machines, and containing $16 \cdot 10^5$ rows in a Parquet file; consider you have a query over an attribute "A = constant" and this attribute contains only 100 different and equiprobable values. Assuming any kind of compression has been disabled, explicit any assumption you need to make and give the amount of raw data (i.e., do not count metadata) it would need to fetch from disk.

- Replication factor: 3 (default)
- Chunk size: 128MB (default)
- Rowgroup size: 32MB

There are

$$\frac{3.2GB}{32MB} = 100 \ RowGroups,$$

each RowGroup has

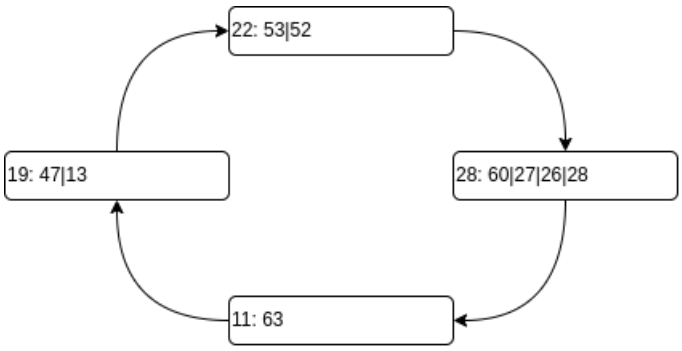
$$16000 \ rows.$$

$$SF = 0.01$$

$$P\left(RGSelected\right) = 1 - (1 - SF)^{Rows(RG)} = 1 - (1 - 0.01)^{16000} = 1 - 0.99^{16000} = 1,$$

so most probably all rowgroups will be fetched from disk. This means we would fetch the whole file from disk.

Exercise 0.3. Given an empty Consistent Hash with $h(x) = x \mod 32$ (i.e., we directly take module 32 to both the keys and the bucket IDs), and unlimited capacity in each bucket, consider you have a cluster of four machines with IDs 19, 22, 75, 92, and draw the result of inserting the following keys in the given order: 12, 4, 10, 49, 42, 60, 63, 53, 47, 27, 26, 28, 13, 52.



Exercise 0.4. Assume you have a MongoDB collection which occupies 6 chunks UNevenly distributed in 3 shards (i.e., 1, 2 and 3 chunks per shard respectively). Being the document Id also the shard key, the chunk of a document is determined by means of a hash function. Assuming that accessing one document takes one time unit (existing indexes are used at no cost) and we have 6,000 documents in the collection, k of which have value "YYY" for attribute "other", how many time units would take the following operations[1]:

- FindOne({ id : "XXX"}): in this case, we get the shard by applying the hash to the id. Also, since mongoDB maintians an index for the keys, we would retrieve the document in 1 unit of time.
- Find({ id : {$in : [1, .., 3000]}}), being [1,..,6000] the range of existing IDs: in this case, the best approach is to query the three shards with the 3000 keys, because doing this one key at a time would entail a high overhead. If we assume equally likely keys, then we would find 500 in the first shard, 1000 in the second shard and 1500 in the third shard. The time would then be 1500 units (since this operation is done in parallel).
- Find({other : "Y Y Y "}), being the attribute indexed. Since the attribute is indexed, we know exactly the documents that need to be retrieved. Therefore, the time would be $\max\{f \cdot 1000, f \cdot 2000, f \cdot 3000\} = f \cdot 3000$ units where $f = \frac{k}{6000}$.
- Find({other : "Y Y Y "}), being the attribute NOT indexed. In this case, all records must be checked, so the time is 3000 units (checking all documents in the bigger shard).

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

```
Employees.txt with fields:
  EmployeeID; EmployeeName; YearlySalary; CityOfResidence; SiteOfWork

  EMP4;RICARDO;250000;MADRID;DPT4
  EMP5;EULALIA;150000;BARCELONA;DPT5
  EMP6;MIQUEL;125000;BADALONA;DPT5
  EMP7;MARIA;175000;MADRID;DPT6
  EMP8;ESTEBAN;150000;MADRID;DPT6
  ...


Departments.txt with fields:
  SiteID; DepartmentName; StreetNumber; StreetName; City

  DPT1;DIRECCIO;10;PAU CLARIS;BARCELONA
  DPT2;DIRECCIO;8;RIOS ROSAS;MADRID
  DPT3;MARKETING;1;PAU CLARIS;BARCELONA
  DPT4;MARKETING;3;RIOS ROSAS;MADRID
  ...
```

Consider the following PySpark code and answer the questions below:

```
source1 = spark.read.format("csv").load("employees.txt", header='false', inferSchema='true',
    sep=";")
source2 = spark.read.format("csv").load("departments.txt", header='false', inferSchema='true',
     sep=";")
A = source1.toDF("eID","eName","eSalary","eCity","eDpt","eProj")
B = source2.toDF("dID","dArea","dNumber","dStreet","dCity")
C = A.select(A.eCity.alias("city"))
D = B.select("dArea")
E = D.crossJoin(C)
F = B.select("dArea",B.dCity.alias("city"))
G = E.subtract(F)
H = G.select("dArea")
result = D.subtract(H)
```

1. **State in natural language the corresponding query it would answer.** *

   Get all department areas located in a city in which at least one employee works, even in a different department.

[1] As typically in RDBMS optimizers, assume uniform distribution of values and statistical independence between pairs of attributes.

2. **Clearly indicate any mistake or improvement you can fix/make in the code. For each of them give (1) the line number, (2) pseudo-code to implement the fix, and (3) brief rationale.**

   (a) There is no action, so we could add it at the end of the computations.

   (b) Line 3 and 4: Instead of reading the CSV file without headers and then renaming the columns, we can directly provide the schema while reading the file. This will make the code cleaner and improve performance as well.

```
schema1 = StructType([
    StructField("eID", StringType()),
    StructField("eName", StringType()),
    StructField("eSalary", FloatType()),
    StructField("eCity", StringType()),
    StructField("eDpt", StringType())
])
schema2 = StructType([
    StructField("dID", StringType()),
    StructField("dArea", StringType()),
    StructField("dNumber", IntegerType()),
    StructField("dStreet", StringType()),
    StructField("dCity", StringType())
])
A = spark.read.format("csv").schema(schema1).load("employees.txt", sep=";")
B = spark.read.format("csv").schema(schema2).load("departments.txt", sep=";")
```

   (c) Line 10: The subtract function is an expensive operation as it requires a full shuffle of the data. Instead, we can perform a left anti-join operation to achieve the same result but in a more efficient way.

```
H = E.join(F, on=['dArea', 'city'], how='left_anti')
```

   (d) Finally, the use of crossJoin in Line 9 can lead to very large intermediate data and should be avoided if possible. A more effective method would be to join the data on the common attribute 'city' instead. This would give us the department areas that are located in the cities where at least one employee resides, without generating all possible combinations first.

```
E = D.join(C, D.dArea == C.city)
```