Data Science and Database Technologies

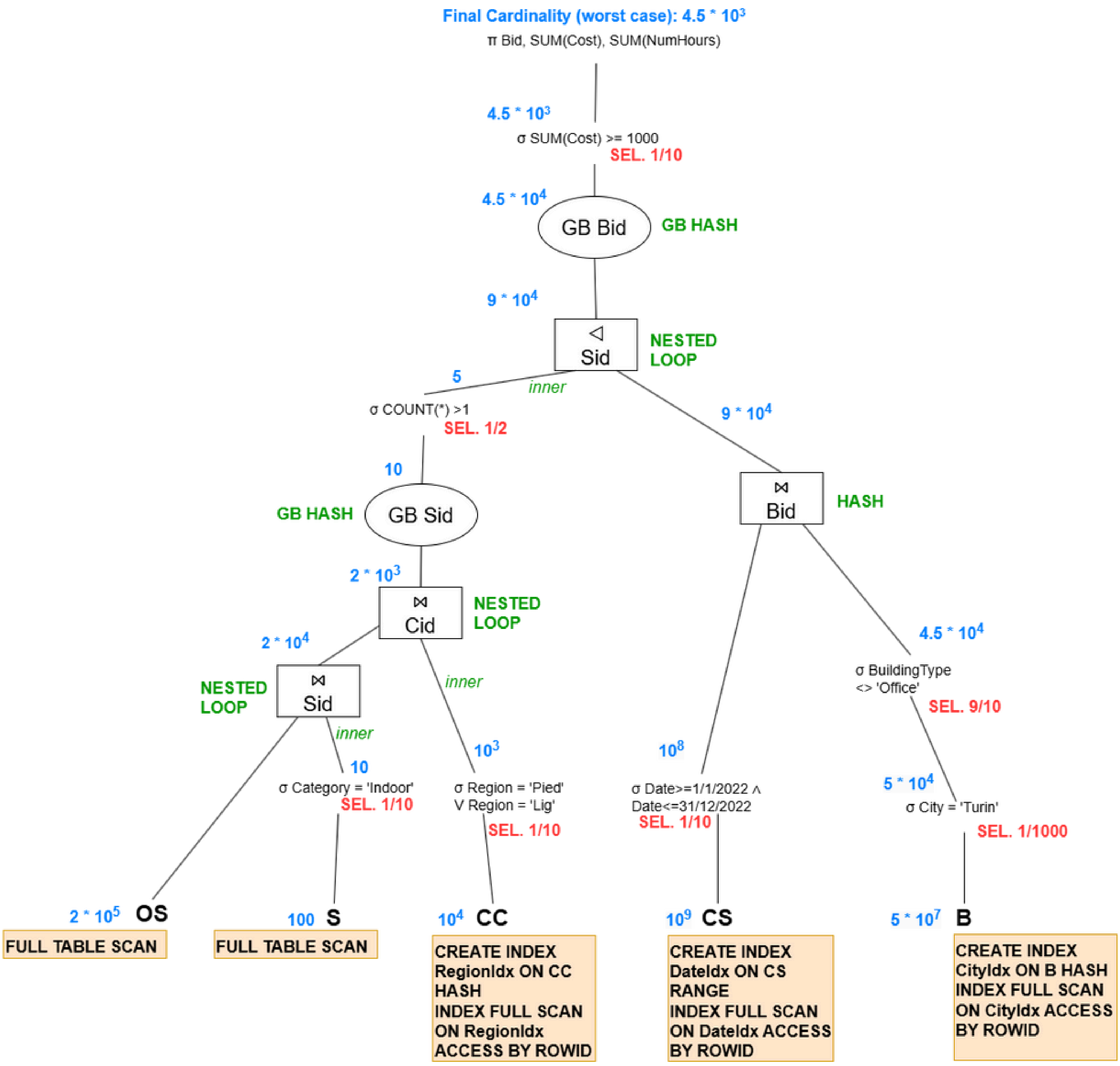
HOMEWORK #3 - Revalor Riccardo - s339423

Question N.1: *Report the corresponding algebraic expression and specify the cardinality of each node (representing an intermediate result or a leaf). If necessary, assume a data distribution. Also analyze the GROUP BY anticipation.*

The **IN** clause specifies a semijoin (◁) operation.
I had to discuss the different join orders and choose the one minimizing the carrdinality of intermediate results. Some different join orders analyzed are:

1. { [(OS ⋈ CC) ⋈ S] ◁ CS } ⋈ B
2. [(OS ⋈ CC) ⋈ S] ◁ (CS ⋈ B)
3. [(OS ⋈ S) ⋈ CC] ◁ (CS ⋈ B)

At the end I verified that option 2 and 3 are equivalent and both minimize the intermediate results, so I'll report just the third one:


Final Cardinality (worst case): 4.5 * 10^3
π Bid, SUM(Cost), SUM(NumHours)
4.5 * 10^3
σ SUM(Cost) >= 1000
SEL. 1/10
4.5 * 10^4
GB Bid
GB HASH
9 * 10^4
◁ Sid
NESTED LOOP
5
inner
σ COUNT(*) >1
SEL. 1/2
9 * 10^4
10
GB HASH
GB Sid
⋈ Bid
HASH
2 * 10^3
⋈ Cid
NESTED LOOP
2 * 10^4
4.5 * 10^4
inner
σ BuildingType <> 'Office'
NESTED LOOP
⋈ Sid
SEL. 9/10
inner
10^3
10^8
10
5 * 10^4
σ Category = 'Indoor'
SEL. 1/10
σ Region = 'Pied' V Region = 'Lig'
SEL. 1/10
σ Date>=1/1/2022 ∧ Date<=31/12/2022
SEL. 1/10
σ City = 'Turin'
SEL. 1/1000
2 * 10^5 OS
FULL TABLE SCAN
100 S
FULL TABLE SCAN
10^4 CC
CREATE INDEX RegionIdx ON CC HASH INDEX FULL SCAN ON RegionIdx ACCESS BY ROWID
10^9 CS
CREATE INDEX DateIdx ON CS RANGE INDEX FULL SCAN ON DateIdx ACCESS BY ROWID
5 * 10^7 B
CREATE INDEX CityIdx ON B HASH INDEX FULL SCAN ON CityIdx ACCESS BY ROWID

<u>**Explanation of the most relevant cardinalities:**</u>

Data distribution assumed: **Uniform distribution**.

- **Join #1 (OS ⋈ S):** The attribute Sid is the primary key of table S and a foreign key of table OS. Since the reduction factor os S is 1/10, the result of this join is composed of at most 2 x 10 ^5 x 1/10 = 2 x 10^4 tuples.
- **Join #2 (OS ⋈ S)] ⋈ CC:** Table CC has got Cid as primary key, so I have to multiply the result of the previous join (Join #1) by the Reduction Factor of the left branch which is 1/10. 2 x 10^4.
- **Join #3 (CS ⋈ B):** Bid is the primary key of table B so the resulting tuples after the join are 10^8 x 1/1000 x 9/10 (which is the Reduction Factor of the right table) = 9 x 10^4. Semijoin: this just check wether the right table has some Sid stored in the left table which is

composed of just 5 tuples. Considering the worst case scenario, every tuple has a compatible Sid so we have at most 9 x 10.

- **Semijoin**: the semijoin (so, the IN clause) selects all the tuples of the right table having a Sid belonging to the 5 tuples of the table in the left branch. Assuming a worst-case scenario, all the 9 x 10^4 tuples have an acceptable Sid and are all taken.
- **GB Sid:** Sid is the primary key of S, since S has 100 distinct tuples we have 100 distinct values of Sid at the beginning, but because we apply the selection based on category == 'Indoor', the distinct value of Sid become, in the worst case, 10. As a result, the GB Sid returns at most 10 distinct tuples.
- **HAVING Count(*) >= 1:** the text says the selectivity of this filter is ½, so it returns ½ x 10 tuples = 5 tuples.
- **GB Bid:** Bid is the primary key of table B, which at teh beginning stores 5 x10^7 distinct tuples. Since the total Reduction Factor for Bid is 10^(-3) x 9/10 = 0.0009, at the end we have, in the worst-case scenario, 4.5 * 10^4 distinct Bid, so the GB returns at most 4.5 * 10^4 distinct tuples.

<u>**GB Anticipation**</u>

We have 2 GB:

1) **GB Sid (+ HAVING clause):** surely I cannot anticipate it on the left branch since table CC does not contain Sid as attribute. Regarding its possible anticipation on the right branch, If I do that I'll lose attribute Cid and I will not be able to perform the subsequent Join by Cid. If I anticipate a GB Sid, Cid (+ HAVING Clause), I'll mess up the groupings and change their size.

2) **GB Bid (+ HAVING clause):** it's the same as the previous GB, the only way would be to ancitipate it on the the right branch (containing tuples derived from table B which has Bid as primary key), bu I would end up loosing Sid attribute, not being able to perform te subsequent join. If I anticipate a GB Sid, Bid (+ HAVING Clause), I'll mess up the groupings and change their size.

So, long story short, no GB anticipations!

Question N.2: *Select one or more secondary physical structures to increase query performance. Justify your choice and report the corresponding execution plan (join orders, access methods, etc.).*

- **Table S:** small table, no indexes → `FULL TABLE SCAN`
- **Table OS:** no selections/filters → `FULL TABLE SCAN`
- **Table CC:** it's a big table (> 10^3), so creating indices would be convenient! Also, the selection has a selectivity of 1/10 (at the limit but still acceptable as a good selectivity) → I can create an **index on attribute Region!** Since it's not the primary key and after I'll need to retrieve primary Cid to perform a join, the index is ***not covering*** and is ***unclustered/secondary.*** *I choose to create an* ***Hash*** *index since Region is a categorical attribute:*
  ```
  CREATE INDEX RegionIdx ON CC HASH
  INDEX FULL SCAN ON RegionIdx ACCESS BY ROWID
  ```
- **Table CS:** it's a very big table and the selection has a selectivity of 1/10, so I can create an **index on Date**! Date is not the primary key so I'll eventually have to access to Bid for the subsequent join, so the index is ***not covering*** and is ***unclustered/secondary***. Since a range of dates is specified, I can use a **B+Tree index**:

```
CREATE INDEX DateIdx ON CS RANGE
INDEX FULL SCAN ON DateIdx ACCESS BY ROWID
```

- **Table B:** it’s a big table. The selection based on BuildingType has a poor selectivity (9/10 >> 1/10, it’s almost 1, very poor!), so I don’t create an index fo that. On the other hand, I can create an **index on City**! City is not the primary key so I’ll eventually have to access to Bid for the subsequent join, so the index is ***not covering*** and is ***unclustered/secondary***. I create and **Hash** index since City is a categorical attribute.

```
CREATE INDEX CityIdx ON B HASH
INDEX FULL SCAN ON CityIdx ACCESS BY ROWID
```

Join Types:

- **Join #1 (OS ⋈ S):** Nested Loop since we have a small table (which is also the inner table, table S) and a big one (table OS which is also the outer table).
- **Join #2 (OS ⋈ S)] ⋈ CC:** Nested Loop aswell. Table C is the small table and the inner one.
- **Join #3 (CS ⋈ B):** Hash Join since both tha tables are big.
- **Semijoin**: Nested Loop, where the left table (which has just 5 tuple, very small!) is the inner table.

GB Types:

- **GB Sid:** GB Hash because the table coming before is big and we don’t already have a Hash function for that.
- **GB Bid:** GB Hash because the table coming before is big and we don’t already have a Hash function for that.