Exercise session 2 : Query Execution

Exercise 1: Query Processing Models Comparison
Explain the key differences between the following query processing models:
- Iterator (Tuple-at-a-time)
- Block-oriented (Column-at-a-time)
- Vectorization

The iterator model processes queries one tuple (row) at a time. Operators expose a next() function, which produces one element at a time. Each operator (e.g., a join or filter) requests tuples from its child operators recursively.
It is very simple and doesn't require massive amounts of resources, and has a high function call overhead due to repeated calls to next().

The block oriented model processes data in blocks (or batches), often column-wise.
Operators return blocks directly (e.g., arrays or chunks of rows) instead of single tuples. The next() calls return all information needed for further processing in one go. This reduces call overheads but requires a lot more hardware resources.

Vectorization is somewhat in between: next() calls return batches of data (e.g. 32 rows, not the whole table and not 1 row).

Given the SQL query below, give one possible plan for execution and describe how each of these models would execute your plan.

```
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
  AND B.c = B.d
  AND B.value > 100;
```

We propose the following plan:
- *The first step would be performing a nested loop join or a hash join between tables A and B on the condition A.id = B.id.*
- *After the join, the rows that satisfy the condition B.c = B.d and B.value > 100 are filtered.*
- *Finally, the required columns A.id and B.value are selected for the output.*

In relational algebra terms, we pick the following plan:

$$\pi(A.id, B.value)\ (\sigma(B.c = B.d \wedge B.value > 100)\ (A \bowtie\_id\ B))$$

For the iterator model:
For each tuple in A, the system would look for the first matching tuple in B where A.id = B.id, possibly using a hash table we previously built. We transmit that to the filter operator. It checks if B.c = B.d and B.value > 100. If these conditions are met, it emits the result to the output. The next call to next() has to pick up the work where the last call returned.

For the block-oriented model, we will first generate the entire joined table (A ⋈ (A.id = B.id) B), then filter it, then project it.

Vectorization looks similar to the iterator model description, but it processes rows a batch at a time.

Which model would have the best performance if A and B both have billions of elements? Why?

Vectorization has a better performance than iterator (which is critical in large operations) and a more manageable memory cost than block-oriented (also critical).

Exercise 2: Implementing Query Operators in the Iterator Model

Consider the following class

```
class Operator:
   def next(self):
      pass #todo implement this in subclasses
```

Provide pseudo-code that defines the Project, Select and Join operators in the Iterator model.
Hint: Use constructors to store state that should be maintained across different next() calls, like the Select predicate or the Join key.
Hint 2: You may use dictionaries for the Join operator. Assume that all joins are equi-joins on two relations using a single attribute from each relation as the join attribute.

```
class Project:
   self.projection = … # constructor defined
   self.inner = …
   def next():
       Return self.projection(self.inner.next())

class Select:
   self.predicate = … # constructor defined
   self.inner = …
   def next():
       elem = self.inner.next()
       while(elem is not None and !self.predicate(elem)):
          elem = self.inner.next()
       return elem

class Join:
   self.table1, self.table2 = … #constructor defined
   self.hashtable = self.generate_hash_table()
   def next():
      elem = self.table1.next()
      while elem is not None:
         if row in self.hashtable:
            return self.hashtable[row]
         elem = self.table1.next()

   def generate_hash_table():
      elem = self.table2.next()
      table = {}
      while elem is not None:
         table[elem[id]] = elem[rest] # ‘rest’ is every key in table 2 except for ‘id’
         elem = self.table2.next()
      return table
```

Note : this Join implementation (boldly) assumes self.hashtable (and by extension table2) only has one matching element for a given tuple from table 1. There are several ways to fix this, for example:
We remember an extra integer as a member of the Join instance. It tells us how far we are in the hashtable value for the current table 1 entry (which is now a list of tuples instead of a tuple). We

bump this integer at every next() call. When the integer surpasses the size of the list, we set it back to 0 and we start working on the next table1 entry.

Write a short explanation of the inefficiencies of this approach and how they could be mitigated using a different processing model.

The problem is that we have way too many next() calls, which is slow. The block processing model removes this issue, essentially calling next() once per operator.

In one sentence and at a very high level, what would you change to make it Block-oriented (with a column granularity)?

next() calls should return the entire processed output at once.

Exercise 3: First look at performance problems
Consider the following SQL query:

SELECT E.Name, D.Budget
FROM Employees E, Departments D
WHERE E.department_id = D.id
AND D.Budget > 1000000

Employees has 10,000 rows and Departments has 20.

Give two ways to execute this query using the Select, Join and Project operators.

Which is likely to be more resource-efficient? Why?
Plan 1:

1. *J = Join(Employees, Departments)*
2. *Select(Budget > 1000000) on J*
3. *Project(Name, Budget)*

The join processes all rows of both tables.

In relational algebra notation, this corresponds to
π_Name,Budget (σ_Budget>1000000 (Employees⋈Departments)).

Plan 2:

1. *S = Select(Budget > 1000000) on D*
2. *Join(Employees, S)*
3. *Project(Name, Budget)*

The join only works on the filtered (smaller) Departments table

In relational algebra, π_Name,Budget (Employees⋈(σ_Budget>1000000 (Departments)))

In plan 2, we filter the Departments table first to include only departments with a budget greater than 1 million. This reduces the number of rows involved in the subsequent join. Since Departments has 20 rows, and after applying the filter, we might only get a few rows (depending on how many departments have a budget over 1 million). The join will be performed with fewer or the same amount of rows as plan 1, which can only be more efficient CPU- and memory-wise.