1. **Create a selection query whose chosen plan is a file scan.**

   Query : select * from takes

   Chosen plan: Sequential file scan

   Reasoning behind constructing the query: All the rows have to be retrieved , hence best way is to scan the blocks sequentially and get the records

   Observation: Query time : 975 ms

2. **Create a selection query with and AND of two predicates, whose chosen plan uses an index scan on one of the predicates. You can create indices on appropriate relation attributes to create such a case.**

   Query : create index on takes(course_id)

   select * from takes where course_id = '403' and grade = 'B+'

   Chosen plan: Index scan on course_id

   Reasoning behind constructing the query: Created index on course_id of takes relation so that it takes the index while scanning.Also since the relation is huge, it won't opt for sequential scan.

   Observation: Query time : 11ms, no rows were retrieved

3. **Create a query where PostgreSQL chooses a (plain) index nested loops join (NOTE: the nested loops operator has 2 children. The first child is the outer input, and it may have an index scan or anything else, that is irrelevant. The second child must have an index scan or bitmap index scan, using an attribute from the first child.)**

   Query : create index on takes(course_id)

   select student.dept_name from student,takes where takes.course_id = student.id and takes.course_id = '942'

   Chosen plan: Index scan on student primary key (id) on student

   Index only scan using the course_id index on takes

   Reasoning behind constructing the query: Since the index was constructed on course_id of takes relation, it uses the same to check the condition and search for the

relevant record , student relation already has a primary key index so it uses the same for outer input.

Observation: Query time 12ms, no rows fetched as condition wasn't a relation

4. **Create an index as below, and see the time taken:**
   create index i1 on takes(id, semester, year);
   Similarly see how long it takes to drop the above index using:
   drop index i1;

   Query : Already given

   Chosen plan: NA

   Reasoning behind constructing the query: NA

   Observation: Index creation time : 112 ms

   Index drop time : 13 ms

5. **Create a table takes2 with the same schema as takes but no primary keys or foreign keys. Find how long it takes the execute the query**
   insert into takes2 select * from takes
   Also, find the query plan for the above insert statement.

   Query : Given

   Chosen plan: Insertion on takes2 takes place via sequential scan on takes

   Reasoning behind constructing the query: NA

   Observation: Time taken = 133ms

6. **Next drop the table takes2 (and its rows, as a result), and create it again, but this time with a primary key. Run the insert again and measure how long it takes to run. Give its query plan, and explain why the time taken is different this time.**

   Query : Given

   Chosen plan: Insertion on takes2 takes place via sequential scan on takes

   Reasoning : Time is more here because of extra work done to maintain indices structure while insertion. It has to follow all the rules of B+ tree insertion

   Observation: Drop time : 82 ms , insertion time = 194 ms

7. **Consider the following nested subquery:**
   select count(*) from course c where exists (select * from takes t where t.course_id < c.course_id)
   What is the plan is chosen by PostgreSQL. Report the plan and actual execution costs. Explain what is happening.

   Cost : Aggregate (cost=68.04..68.05 rows=1 width=8)

   Nested Loop Semi Join (cost=0.29..67.87 rows=67 width=0)

   Seq Scan on course c (cost=0.00..4.00 rows=200 width=4)

   Index Only Scan using takes_pkey on takes t (cost=0.29..333.57 rows=10000)

   Chosen plan: Aggregate -> Nested loop semi join , sequential scan on course c and index only scan on takes using primary key on takes t,index condition is course_id < c.course_id

   Reasoning : The semi join only takes the keys of the relation course into account if these are also present in the associated table Takes.In postgres a multicolumn B+-tree index can be used with query conditions that involve any subset of the index's columns, Here, in our query,the optimiser assumes that course_id index scan on takes is faster ,so it uses this optimisation to perform comparison and then do a nested loop semi join before aggregation.

   Observation: Execution of query time : 31 ms

8. **As above, but with the query s*elect count(*) from course c where c.course_id in (select course_id from takes t)***

   Query : Given

   Chosen plan: Aggregate->Sequential scan on takes t and course c

   Reasoning: Here the aggregation is made after doing sequential scan as all the tuples are to be fetched from takes to check existence, in contrast to the previous query which was conditional.

   Observation: Execution of query time : 13 ms