1. **Create a query where PostgreSQL uses bitmap index scan on relation takes. Explain why PostgreSQL may have chosen this plan.**

Query: select * from student natural join takes where ID = '24932';

Query Plan:

```
Nested Loop  (cost=4.68..61.29 rows=15 width=43)
  ->  Index Scan using student_pkey on student  (cost=0.28..8.29 rows=1 width=24)
        Index Cond: ((id)::text = '24932'::text)
  ->  Bitmap Heap Scan on takes  (cost=4.40..52.84 rows=15 width=24)
        Recheck Cond: ((id)::text = '24932'::text)
        ->  Bitmap Index Scan on takes_pkey  (cost=0.00..4.40 rows=15 width=0)
              Index Cond: ((id)::text = '24932'::text)
```

Reason:

- Here Bitmap index is made on the takes as id is not a primary key here. Also in student, id is primary so , index scan is feasible here.

2. **Create a selection query with an 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: select * from student natural join takes where ID = '24932' and semester='Fall'

Query Plan:

```
Nested Loop  (cost=4.72..37.16 rows=7 width=43)
  ->  Index Scan using student_pkey on student  (cost=0.28..8.29 rows=1 width=24)
        Index Cond: ((id)::text = '24932'::text)
  ->  Bitmap Heap Scan on takes  (cost=4.44..28.80 rows=7 width=24)
        Recheck Cond: (((id)::text = '24932'::text) AND ((semester)::text = 'Fall'::text))
        ->  Bitmap Index Scan on takes_pkey  (cost=0.00..4.44 rows=7
width=0)
              Index Cond: (((id)::text = '24932'::text) AND ((semester)::text = 'Fall'::text))
```

Reason:
Here id is primary key in student, so while retrieving the given id , index scan is being executed.

3. **Create a query where PostgreSQL chooses a (plain) index nested loops join (NOTE: PostgreSQL uses nested loops join even for indexed nested loops join. 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: select * from instructor, student where student.id = instructor.id and student.id = '3335';

Chosen Plan:

```
Nested Loop  (cost=0.28..9.93 rows=1 width=55)
  ->  Seq Scan on instructor  (cost=0.00..1.63 rows=1 width=31)
        Filter: ((id)::text = '3335'::text)
  ->  Index Scan using student_pkey on student  (cost=0.28..8.29 rows=1 width=24)
        Index Cond: ((id)::text = '3335'::text)
```

Reason:

- Nested loop operator have two children. Here for first child, seq scan is being done, and for second child, index scan is being done as in second child id the primary key so index scan used to retrieve the given id in few fetches and hence are matched with every tuple of the first child to produce the results. Therefore it meets the requirements of indexed nested loop join.

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

- Time taken to create the index : 305 ms
- Time taken to drop the index : 90 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 – Creating schema takes2:

```
create table takes2
      (ID                   varchar(5),
       course_id            varchar(8),
       sec_id               varchar(8),
```

```
 semester              varchar(6),
 year                  numeric(4,0),
 grade              varchar(2)
);
```

- Time Taken: 145ms.

- **insert into takes2 select * from takes**
- Time Taken – 340ms.

Query Plan :

```
Insert on takes2  (cost=0.00..520.00 rows=0
width=0)
  ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000 width=24)
```

Reason:

Values are inserted one by one without any constraint checking so query is quiet faster as compared to one having some constraint.

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:

```
create table takes2
    (ID                   varchar(5),
     course_id            varchar(8),
     sec_id               varchar(8),
     semester             varchar(6),
     year                 numeric(4,0),
     grade             varchar(2),
     primary key (ID, course_id, sec_id, semester, year)
    );
```

- insert into takes2 select * from takes ;

- Time taken : 654ms

Query Plan:

```
Insert on takes2  (cost=0.00..520.00 rows=0
width=0)
  ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000 width=24)
```

Reason:

Time taken is more here as compared to earlier one as here, primary constraint need to be checked after each insertion where. If any duplicate value of primary key comes then that would be not allowed.

7. **Create a query where PostgreSQL chooses a merge join (hint: use an order by clause).**

Query:

```
select * from takes join takes2 on takes.id= takes2.id where  takes.year='2002'
order by takes.id;
```

Query Plan:

```
Merge Join  (cost=826.50..3844.13 rows=58976 width=48)
  Merge Cond: ((takes2.id)::text = (takes.id)::text)
  ->  Index Scan using takes2_pkey on takes2  (cost=0.29..2058.28 rows=30000 width=24)
  ->  Sort  (cost=826.22..835.91 rows=3879 width=24)
        Sort Key: takes.id
        ->  Seq Scan on takes  (cost=0.00..595.00 rows=3879 width=24)
              Filter: (year = '2002'::numeric)
```

Reason:

- Here no. of entries in take is huge, so it will prefer merge join. Hence will sort both the table and merge them using merge join.

Note: takes2 is the table we created in ques:6

8. **Add a LIMIT 10 ROWS clause at the end of the previous query, and see what algorithm method is used. (LIMIT *n* ensures only *n* rows are output.) Explain what happened, if the join algorithm changes; if the plan does not change, create a different query where the join algorithm changes as a result of adding the LIMIT clause.**

Query:

```
select * from takes  join  takes2 on takes.id= takes2.id where  takes.year='2002'
order by takes.id limit 10;
```

Query Plan :

```
Limit  (cost=0.58..1.30 rows=10 width=48)
  ->  Nested Loop  (cost=0.58..4206.48 rows=58976 width=48)
        ->  Index Scan using takes2_pkey on takes2  (cost=0.29..2058.28 rows=30000 width=24)
        ->  Memoize  (cost=0.30..0.52 rows=2 width=24)
              Cache Key: takes2.id
```

```
Cache Mode: logical
->  Index Scan using takes_pkey on takes  (cost=0.29..0.51 rows=2 width=24)
      Index Cond: (((id)::text = (takes2.id)::text) AND (year = '2002'::numeric))
```

Reason:

Here a limit is there to output 10 tuple only so, instead of going with merge join Here it will do the index scan and seq scan as time is saved instead of sorting all entries. The moment query got 10 tuple as ouput it will stop.

9. **Create an aggregation query where PostgreSQL uses (in-memory) hash-aggregation.**

Query: select year, count(*) from takes group by year;

QUERY PLAN:

```
HashAggregate (cost=670.00..670.10 rows=10 width=13)
  Group Key: year
  ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000
width=5)
```

Reason:

Hash aggregate is being as used as year have few distinct value. So, all the tuple can be mapped to few buckets . Hence cost will be less here as compared to any other aggregation.

10. **Create an aggregation query where PostgreSQL uses sort-based aggregation**

Query:

select sum(salary) from instructor group by id order by id;

Query Plan:

```
GroupAggregate  (cost=2.91..3.91 rows=50 width=37)
  Group Key: id
  ->  Sort  (cost=2.91..3.04 rows=50 width=14)
        Sort Key: id
        ->  Seq Scan on instructor  (cost=0.00..1.50 rows=50 width=14)
```

Reason:

Here, order by id is used in the query which sort the tuple on the basis of the id. Hence, sort aggregation is used which sort the tuples first and then aggregate. So ,no need to sort after the aggregate, as it would be

sorted itself. Also the aggregate function tuple will be present at one place as the tuple are sorted that will reduce the cost as compared to the hash aggregation.