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

Solution:
Here's an example query that could use a bitmap index scan:

```
SELECT *
FROM takes
WHERE course_id = 'CS101';
```

PostgreSQL may choose a bitmap index scan in this case because it is an efficient way to search for values in a large table using an index.

When the query is executed, PostgreSQL can use the bitmap index to create a bitmap (a series of 1s and 0s) indicating which rows in the table match the query condition (student_id = 123). The bitmap index scan retrieves this bitmap directly from the index and uses it to read only the necessary data pages from the table. This can be much faster than scanning the entire table.
A bitmap index scan is particularly useful when the query condition matches a relatively small fraction of the rows in the table. In such cases, the overhead of using the index and the bitmap can be more than offset by the savings in I/O and CPU time.

```
explain SELECT * FROM takes WHERE course_id = 'CS101';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on takes  (cost=0.00..595.00 rows=1 width=24)
   Filter: ((course_id)::text = 'CS101'::text)
(2 rows)


explain analyze SELECT * FROM takes WHERE course_id = 'CS101';
                                                 QUERY PLAN
-------------------------------------------------------------------------------
-------------------------
 Seq Scan on takes  (cost=0.00..595.00 rows=1 width=24) (actual
time=7.606..7.607 rows=0 loops=1)
   Filter: ((course_id)::text = 'CS101'::text)
   Rows Removed by Filter: 30000
 Planning Time: 0.192 ms
 Execution Time: 7.641 ms
(5 rows)
```

Q)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.

Solution:
Here's an example selection query with an AND of two predicates that uses an index scan on one of the predicates:

```
SELECT *
FROM takes
WHERE ID = 'S01' AND year = 2022;
```

When this query is executed, PostgreSQL will use the index scan on the year attribute to find the matching rows with the specified year value. Then, it will use the index on the primary key to filter out the rows that don't match the specified ID value.

```
explain SELECT * FROM takes WHERE ID = 'S01' AND year = 2022;
                        QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using takes_pkey on takes  (cost=0.29..8.45 rows=1 width=24)
   Index Cond: (((id)::text = 'S01'::text) AND (year = '2022'::numeric))
(2 rows)
```

```
explain analyze SELECT * FROM takes WHERE ID = 'S01' AND year = 2022;
                                                        QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------
 Index Scan using takes_pkey on takes  (cost=0.29..8.45 rows=1 width=24)
(actual time=0.092..0.093 rows=0 loops=1)
   Index Cond: (((id)::text = 'S01'::text) AND (year = '2022'::numeric))
 Planning Time: 0.208 ms
 Execution Time: 0.116 ms
```

Q) Create a selection query with an OR of two predicates, whose chosen plan uses an index scan on atleast one or both of the predicates. You can create indices on appropriate relation attributes to create such a case.

Solution:
Here's an example selection query with an OR of two predicates, whose chosen plan uses an index scan on at least one or both of the predicates:

```
    SELECT *
    FROM section
    WHERE building = 'Taylor' OR semester = 'Fall';
```

We can create indices on the age and salary columns to optimize this query. For example, we can create a b-tree index on the age column and another b-tree index on the salary column:

```
explain SELECT * FROM section WHERE building = 'Taylor' OR semester =
'Fall';
                                      QUERY PLAN
------------------------------------------------------------------------
-------------
 Seq Scan on section  (cost=0.00..2.50 rows=58 width=28)
   Filter: (((building)::text = 'Taylor'::text) OR ((semester)::text =
'Fall'::text))
(2 rows)


explain analyze SELECT * FROM section WHERE building = 'Taylor' OR
semester = 'Fall';
                                                QUERY PLAN
------------------------------------------------------------------------
---------------------------
 Seq Scan on section  (cost=0.00..2.50 rows=58 width=28) (actual
time=0.081..0.166 rows=60 loops=1)
   Filter: (((building)::text = 'Taylor'::text) OR ((semester)::text =
'Fall'::text))
   Rows Removed by Filter: 40
 Planning Time: 0.186 ms
 Execution Time: 0.209 ms
```

Q) 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.)

Solution:
Here's an example query where PostgreSQL would choose a plain index nested loops join:

```
    SELECT *
    FROM student s
    JOIN takes t ON s.ID = t.ID
    WHERE s.dept_name = 'Computer Science';
```

```
explain SELECT * FROM student s JOIN takes t ON s.ID = t.ID WHERE
s.dept_name = 'Computer Science';
                                    QUERY PLAN
-------------------------------------------------------------------------
-------
 Nested Loop  (cost=4.40..92.99 rows=15 width=48)
   ->  Seq Scan on student s  (cost=0.00..40.00 rows=1 width=24)
         Filter: ((dept_name)::text = 'Computer Science'::text)
   ->  Bitmap Heap Scan on takes t  (cost=4.40..52.84 rows=15 width=24)
         Recheck Cond: ((id)::text = (s.id)::text)
         ->  Bitmap Index Scan on takes_pkey  (cost=0.00..4.40 rows=15
width=0)
               Index Cond: ((id)::text = (s.id)::text)
```

```
explain analyze SELECT * FROM student s JOIN takes t ON s.ID = t.ID WHERE
s.dept_name = 'Computer Science';
                                                    QUERY PLAN
-------------------------------------------------------------------------
---------------------------------
 Nested Loop  (cost=4.40..92.99 rows=15 width=48) (actual
time=0.528..0.529 rows=0 loops=1)
   ->  Seq Scan on student s  (cost=0.00..40.00 rows=1 width=24) (actual
time=0.527..0.527 rows=0 loops=1)
         Filter: ((dept_name)::text = 'Computer Science'::text)
         Rows Removed by Filter: 2000
   ->  Bitmap Heap Scan on takes t  (cost=4.40..52.84 rows=15 width=24)
(never executed)
         Recheck Cond: ((id)::text = (s.id)::text)
         ->  Bitmap Index Scan on takes_pkey  (cost=0.00..4.40 rows=15
width=0) (never executed)
               Index Cond: ((id)::text = (s.id)::text)
 Planning Time: 0.752 ms
 Execution Time: 0.611 ms
```

Q) 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;
```

Solution:
We use the '\timing' functionality to enable timing. "explain analyze" doesnt work on the create index command;

```
=# \timing

=# create index i1 on takes(id, semester, year);
CREATE INDEX
Time: 67.070 ms

=#DROP INDEX
Time: 2.525 ms
```

Q)Create an empty table takes2 with the same schema and data 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.

Solution:
to create an empty table takes2 with the same schema and data as takes but without any primary keys or foreign keys using the following SQL command:

```
    CREATE TABLE takes2 AS
    SELECT * FROM takes WHERE 1=0;
```

Time to exec: Time: 3.492 ms

```
EXPLAIN CREATE TABLE takes2 AS SELECT * FROM takes WHERE 1=0;
                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=0)
   One-Time Filter: false
(2 rows)

Time: 0.847 ms
```

```
EXPLAIN ANALYZE CREATE TABLE takes2 AS SELECT * FROM takes WHERE 1=0;
                                      QUERY PLAN
-------------------------------------------------------------------------------
-----------
 Result  (cost=0.00..0.00 rows=0 width=0) (actual time=0.002..0.003
rows=0 loops=1)
   One-Time Filter: false
 Planning Time: 0.123 ms
 Execution Time: 1.532 ms
(4 rows)

Time: 2.956 ms
```

```
explain analyze insert into takes2 select * from takes;
                                                      QUERY PLAN
-------------------------------------------------------------------------------
---------------------------------------
 Insert on takes2  (cost=0.00..520.00 rows=0 width=0) (actual
time=33.368..33.368 rows=0 loops=1)
   ->  Seq Scan on takes  (cost=0.00..520.00 rows=30000 width=24) (actual
time=0.010..3.518 rows=30000 loops=1)
 Planning Time: 0.124 ms
 Execution Time: 33.396 ms
(4 rows)

Time: 36.617 ms
```

Q)Next measure the time it takes to modify takes2 by using alter table to add the primary key constraint.

Solution:

```
ALTER TABLE takes2 ADD PRIMARY KEY (id);
    ERROR:  could not create unique index "takes2_pkey"
    DETAIL:  Key (id)=(75510) is duplicated.
    Time: 64.267 ms
```

Q)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. Compare the time taken with the sum of the times for the previous two parts of the question. What do you conclude from the above.

Solution:

```
CREATE TABLE takes2 (
    id integer PRIMARY KEY,
    course_id varchar(20),
    sec_id varchar(10),
    semester varchar(6),
    year numeric(4,0),
    grade varchar(2)
);
```

O/P:

```
    CREATE TABLE
    Time: 4.559 ms
```

followed by

```
INSERT INTO takes2 (id, course_id, sec_id, semester, year, grade)
SELECT id::integer, course_id, sec_id, semester, year, grade FROM takes
WHERE 1=0;
```

(had to do this cause was getting dtype mismatch error)

O/P:

```
    INSERT 0 0
    Time: 0.933 ms
```

Explanation:
Based on the time taken for each of the previous two parts of the question, we can conclude that adding a primary key constraint to an existing table using ALTER TABLE command can take significantly longer than creating a new table with the primary key constraint.

This is because adding a primary key constraint requires PostgreSQL to scan through the entire table and verify that the column(s) used to create the primary key do not contain duplicate values. This operation can be time-consuming, especially for larger tables.

On the other hand, creating a new table with a primary key constraint does not require scanning through an existing table, so it can be relatively faster.

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

Solution:
Here's an example query that will use a merge join:

```
    SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    ORDER BY student.ID;
```

This query joins the student and takes tables and sorts the results by student.ID.
By ordering the results this way, PostgreSQL is more likely to use a merge join to efficiently merge the sorted results from each table.

```
explain SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    ORDER BY student.ID;
                                        QUERY PLAN
-------------------------------------------------------------------------------
---------------
 Merge Join  (cost=0.56..2568.55 rows=30000 width=48)
   Merge Cond: ((student.id)::text = (takes.id)::text)
   ->  Index Scan using student_pkey on student  (cost=0.28..130.27
rows=2000 width=24)
   ->  Index Scan using takes_pkey on takes  (cost=0.29..2058.28
rows=30000 width=24)
(4 rows)
```

```
explain analyze SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    ORDER BY student.ID;
                                                              QUERY PLAN
-------------------------------------------------------------------------------
-------------------------------------------------------------
 Merge Join  (cost=0.56..2568.55 rows=30000 width=48) (actual
time=0.061..34.556 rows=30000 loops=1)
   Merge Cond: ((student.id)::text = (takes.id)::text)
   ->  Index Scan using student_pkey on student  (cost=0.28..130.27
rows=2000 width=24) (actual time=0.016..1.398 rows=2000 loops=1)
   ->  Index Scan using takes_pkey on takes  (cost=0.29..2058.28
rows=30000 width=24) (actual time=0.032..21.017 rows=30000 loops=1)
 Planning Time: 0.689 ms
 Execution Time: 36.418 ms
(6 rows)
```

Q)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.

Solution:

```
    SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    JOIN course ON takes.course_id = course.course_id
    ORDER BY student.ID, takes.semester, takes.year limit 10;
```

Initially was merge-join then went to nested-loop join. Merge join is generally more efficient when joining large tables, as it requires sorting the input tables and then merging the sorted results. However, if one or both of the tables being joined are small, a nested loop join may be more efficient.
The planner may have decided that the overhead of sorting and merging the larger result set using a merge-join was not worth the potential speedup.

```
explain SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    JOIN course ON takes.course_id = course.course_id
    ORDER BY student.ID, takes.semester, takes.year limit 10;

                                                        QUERY PLAN
------------------------------------------------------------------------------
---------------------------------
 Limit  (cost=2.87..4.46 rows=10 width=83)
   ->  Incremental Sort  (cost=2.87..4755.26 rows=30000 width=83)
         Sort Key: student.id, takes.semester, takes.year
         Presorted Key: student.id
         ->  Nested Loop  (cost=0.72..3330.33 rows=30000 width=83)
               ->  Merge Join  (cost=0.56..2568.55 rows=30000 width=48)
                     Merge Cond: ((student.id)::text = (takes.id)::text)
                     ->  Index Scan using student_pkey on student
(cost=0.28..130.27 rows=2000 width=24)
                     ->  Index Scan using takes_pkey on takes
(cost=0.29..2058.28 rows=30000 width=24)
               ->  Memoize  (cost=0.15..0.17 rows=1 width=35)
                     Cache Key: takes.course_id
                     Cache Mode: logical
                     ->  Index Scan using course_pkey on course
(cost=0.14..0.16 rows=1 width=35)
                           Index Cond: ((course_id)::text =
(takes.course_id)::text)
(14 rows)
```

```
explain analyze SELECT *
    FROM student
    JOIN takes ON student.ID = takes.ID
    JOIN course ON takes.course_id = course.course_id
    ORDER BY student.ID, takes.semester, takes.year limit 10;
```

QUERY PLAN

```
------------------------------------------------------------------------
------------------------------------------------------------------------
--
 Limit  (cost=2.87..4.46 rows=10 width=83) (actual time=0.263..0.269
rows=10 loops=1)
   ->  Incremental Sort  (cost=2.87..4755.26 rows=30000 width=83) (actual
time=0.261..0.265 rows=10 loops=1)
         Sort Key: student.id, takes.semester, takes.year
         Presorted Key: student.id
         Full-sort Groups: 1  Sort Method: quicksort  Average Memory:
26kB  Peak Memory: 26kB
         ->  Nested Loop  (cost=0.72..3330.33 rows=30000 width=83)
(actual time=0.069..0.195 rows=14 loops=1)
               ->  Merge Join  (cost=0.56..2568.55 rows=30000 width=48)
(actual time=0.045..0.094 rows=14 loops=1)
                     Merge Cond: ((student.id)::text = (takes.id)::text)
                     ->  Index Scan using student_pkey on student
(cost=0.28..130.27 rows=2000 width=24) (actual time=0.019..0.021 rows=2
loops=1)
                     ->  Index Scan using takes_pkey on takes
(cost=0.29..2058.28 rows=30000 width=24) (actual time=0.010..0.046
rows=14 loops=1)
               ->  Memoize  (cost=0.15..0.17 rows=1 width=35) (actual
time=0.006..0.006 rows=1 loops=14)
                     Cache Key: takes.course_id
                     Cache Mode: logical
                     Hits: 0  Misses: 14  Evictions: 0  Overflows: 0
Memory Usage: 2kB
                     ->  Index Scan using course_pkey on course
(cost=0.14..0.16 rows=1 width=35) (actual time=0.004..0.004 rows=1
loops=14)
                           Index Cond: ((course_id)::text =
(takes.course_id)::text)
 Planning Time: 1.407 ms
 Execution Time: 0.391 ms
(18 rows)
```

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

Solution:
Here's an example query that uses hash-aggregation in PostgreSQL:

```
    SELECT dept_name, AVG(salary) as avg_salary
    FROM instructor
    GROUP BY dept_name;
```

In this query, we are calculating the average salary for each department. To do this, we group the records by the dept_name column using the GROUP BY clause, and then calculate the average salary using the AVG function. Since we are using an aggregation function, PostgreSQL needs to group the records and compute the aggregate values. In this case, PostgreSQL can use hash-aggregation to perform this task more efficiently. The hash-aggregation algorithm works by hashing the grouping keys (in this case, the dept_name column), and then computing the aggregate values for each group in-memory.

```
explain SELECT dept_name, AVG(salary) as avg_salary
FROM instructor
GROUP BY dept_name;
                              QUERY PLAN
---------------------------------------------------------------------
 HashAggregate  (cost=16.60..19.10 rows=200 width=90)
   Group Key: dept_name
   ->  Seq Scan on instructor  (cost=0.00..14.40 rows=440 width=72)
(3 rows)

Time: 0.972 ms


explain analyze SELECT dept_name, AVG(salary) as avg_salary
FROM instructor
GROUP BY dept_name;
                                                       QUERY PLAN
---------------------------------------------------------------------------
---------------------------------------
 HashAggregate  (cost=16.60..19.10 rows=200 width=90) (actual
time=0.115..0.132 rows=17 loops=1)
   Group Key: dept_name
   Batches: 1  Memory Usage: 40kB
   ->  Seq Scan on instructor  (cost=0.00..14.40 rows=440 width=72)
(actual time=0.020..0.027 rows=50 loops=1)
 Planning Time: 0.160 ms
 Execution Time: 0.188 ms
(6 rows)

Time: 0.810 ms
```

Q) Create an aggregation query where PostgreSQL uses sort-based aggregation

Solution:
Here's an example of an aggregation query where PostgreSQL would use sort-based aggregation:

```
    SELECT dept_name, COUNT(*)
FROM instructor
GROUP BY dept_name
ORDER BY COUNT(*) DESC;
```

This query is counting the number of instructors in each department, and then ordering the results by the count in descending order. PostgreSQL will use sort-based aggregation to perform this query, which involves sorting the data by department and then aggregating the counts.

```
explain SELECT dept_name, COUNT(*)
FROM instructor
GROUP BY dept_name
ORDER BY COUNT(*) DESC;
                                  QUERY PLAN
------------------------------------------------------------------------------
-
 Sort  (cost=26.24..26.74 rows=200 width=66)
   Sort Key: (count(*)) DESC
   ->  HashAggregate  (cost=16.60..18.60 rows=200 width=66)
         Group Key: dept_name
         ->  Seq Scan on instructor  (cost=0.00..14.40 rows=440 width=58)
(5 rows)


explain analyze SELECT dept_name, COUNT(*)
FROM instructor
GROUP BY dept_name
ORDER BY COUNT(*) DESC;
                                                          QUERY PLAN
------------------------------------------------------------------------------
-----------------------------------------------
 Sort  (cost=26.24..26.74 rows=200 width=66) (actual time=0.169..0.173
rows=17 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  HashAggregate  (cost=16.60..18.60 rows=200 width=66) (actual
time=0.067..0.074 rows=17 loops=1)
         Group Key: dept_name
         Batches: 1  Memory Usage: 40kB
         ->  Seq Scan on instructor  (cost=0.00..14.40 rows=440 width=58)
(actual time=0.011..0.018 rows=50 loops=1)
 Planning Time: 0.173 ms
 Execution Time: 0.234 ms
(9 rows)
```