Implementation Techniques for Relational Database Systems CS631

Assignment 01 on PostgreSQL

Question 1)

For a simple query like SELECT * FROM takes; PostgreSQL used sequential scan.
But for example, for this query

select * from takes where Id > '20000' and Id < '30000'; It used bitmap Index scan.

The reason can be following:
Bitmap index scan is like a sequential scan where large data is easier to read and line and like an index scan knowing what data exactly it needs to read.

Question 2)

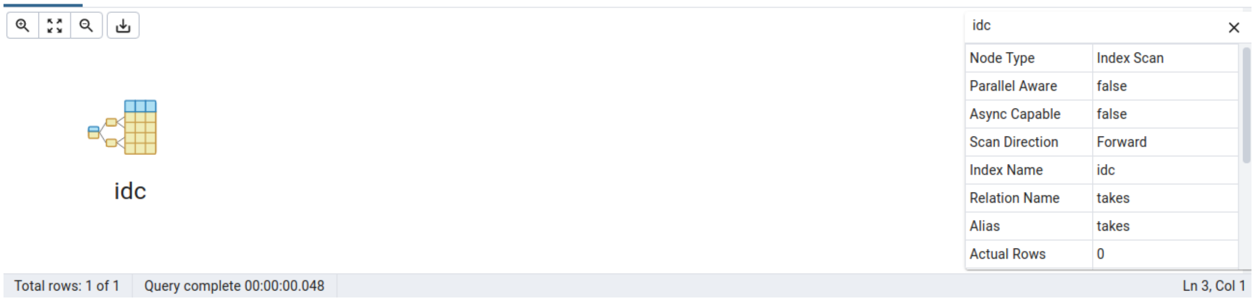
Creating a selection query on AND with two predicates, one of which uses index scan.
So first we generated/created an index on (Id,Course_id) with this query :-

create index idc on takes(Id,course_id);

Then we give this query

select * from takes where course_id='401' and Id = '40000';

earlier it used to sequential scan but now it is giving index scan because we have generated an index on it.



idc

Node Type	Index Scan
Parallel Aware	false
Async Capable	false
Scan Direction	Forward
Index Name	idc
Relation Name	takes
Alias	takes
Actual Rows	0

Total rows: 1 of 1 Query complete 00:00:00.048 Ln 3, Col 1

Question 3)

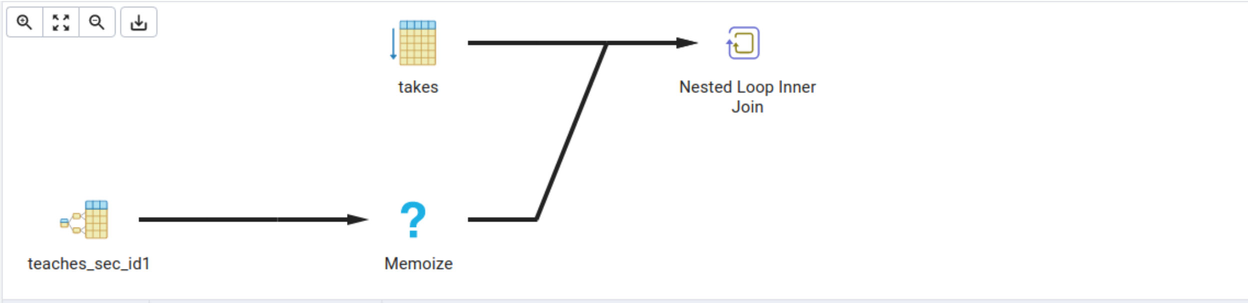
Indexed nested loop join query.

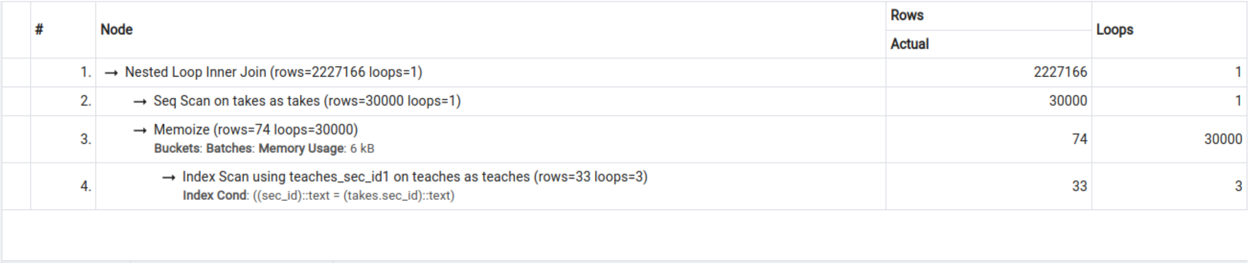
Query Used : select * from takes,teaches where takes.sec_id = teaches.sec_id ;

Result:

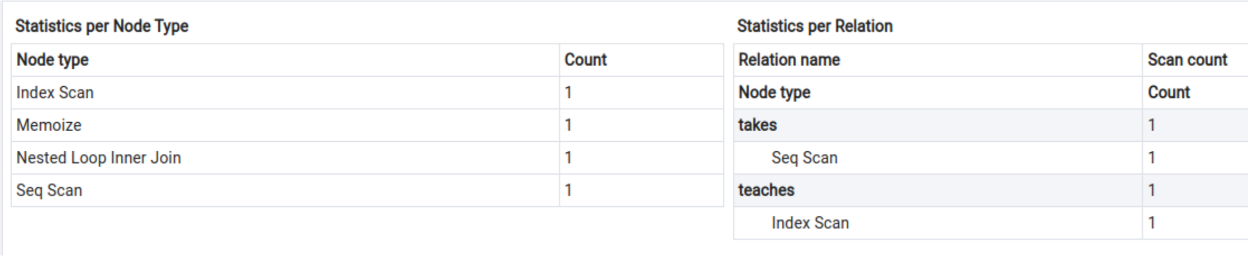
```
Successfully run. Total query runtime: 264 msec.
1 rows affected.
```

Evaluation Plan:





#	Node	Rows Actual	Loops
1.	→ Nested Loop Inner Join (rows=2227166 loops=1)	2227166	1
2.	→ Seq Scan on takes as takes (rows=30000 loops=1)	30000	1
3.	→ Memoize (rows=74 loops=30000) **Buckets**: **Batches**: **Memory Usage**: 6 kB	74	30000
4.	→ Index Scan using teaches_sec_id1 on teaches as teaches (rows=33 loops=3) **Index Cond**: ((sec_id)::text = (takes.sec_id)::text)	33	3



Statistics per Node Type

Node type	Count
Index Scan	1
Memoize	1
Nested Loop Inner Join	1
Seq Scan	1

Statistics per Relation

Relation name	Scan count
Node type	**Count**
takes	1
Seq Scan	1
teaches	1
Index Scan	1

Here we used teaches and takes relations where the indexing is done on the sec_id attribute of the takes relation and takes is the larger relation. Our query plan took teaches (smaller) as outer relation and takes as inner relation. For every outer tuple one index scan is happening in the indexed inner takes relation.

Question 4)

Given query is create index i1 on takes(id, semester, year) and following is the output:

```
Query returned successfully in 229 msec.
```

Similarly for the drop query drop index i1;

```
Query returned successfully in 72 msec.
```

Question 5)

Table creation with no primary or foreign keys with following query:

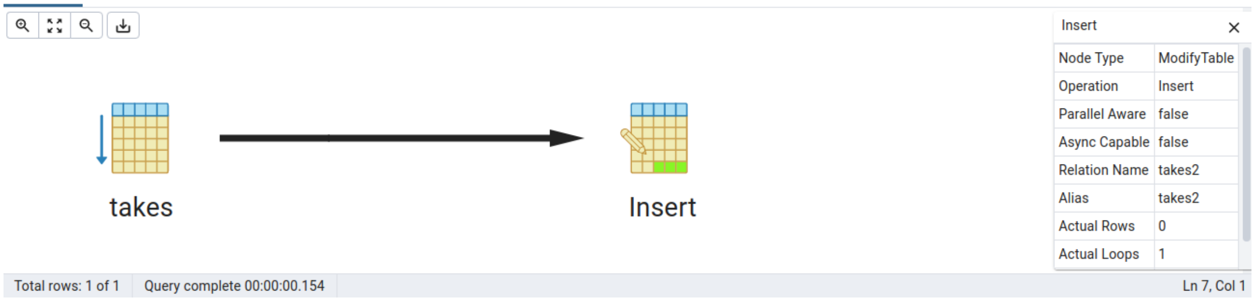
```
create table takes2
	(ID			varchar(5),
	 course_id		varchar(8),
	 sec_id			varchar(8),
	 semester		varchar(6),
	 year			numeric(4,0),
	 grade		varchar(2)
	);
```

Result: CREATE TABLE
Query returned successfully in 92 msec.

Insertion into this table takes2 from table takes:
insert into takes2 select * from takes
Result:		INSERT 0 30000
Query returned successfully in 99 msec.

Query plan used here is as follows:

Insert on takes2 as takes2 with one loop and sequential scans on takes as takes with one loop and 3000 rows affected:



Insert

Node Type	ModifyTable
Operation	Insert
Parallel Aware	false
Async Capable	false
Relation Name	takes2
Alias	takes2
Actual Rows	0
Actual Loops	1

Total rows: 1 of 1	Query complete 00:00:00.154	Ln 7, Col 1

Question 6)

Dropped the table:

drop table takes2;

Result : DROP TABLE

Query returned successfully in 70 msec.

Created new one with primary key:

Query: create table takes

```
(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),
);
```

Result : CREATE TABLE

Query returned successfully in 55 msec.

Adding data in the new one takes2 from takes:

Query: insert into takes2 select * from takes;

Result : INSERT 0 30000

Query returned successfully in 227 msec.

Question 7)

Creating a PostgreSQL query that chooses merge join:

First creating the index on two large tables student (2000 entries) and takes (30000 entries)

Query:

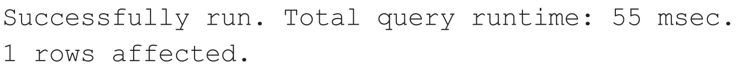
create index studid_idx on student(id);

create index takesid_idx on takes(id);

Now selecting the tuple based on the column where the indexing is done and then order by.

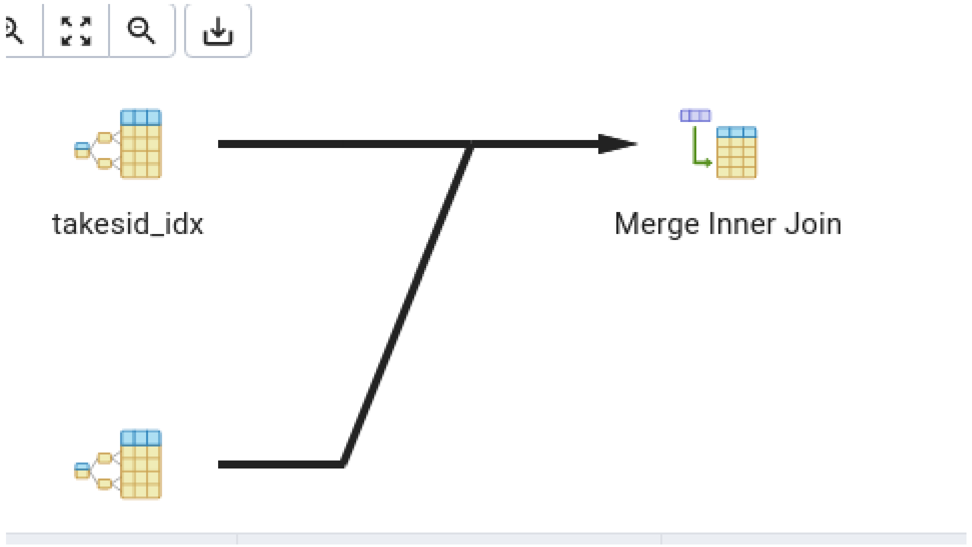
select * from takes join student on takes.id = student.id order by student.id ;

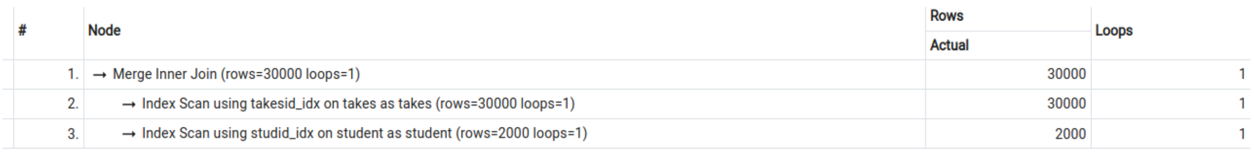
Result:



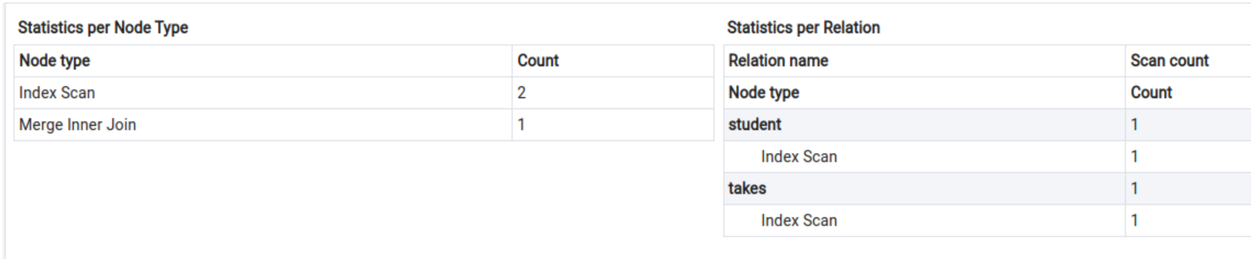
```
Successfully run. Total query runtime: 55 msec.
1 rows affected.
```

Execution plan:





#	Node	Rows	Loops
		Actual	
1.	→ Merge Inner Join (rows=30000 loops=1)	30000	1
2.	→ Index Scan using takesid_idx on takes as takes (rows=30000 loops=1)	30000	1
3.	→ Index Scan using studid_idx on student as student (rows=2000 loops=1)	2000	1



Statistics per Node Type

Node type	Count
Index Scan	2
Merge Inner Join	1

Statistics per Relation

Relation name	Scan count
Node type	Count
student	1
Index Scan	1
takes	1
Index Scan	1

An optimizer chooses Merge join if the relations are too big to fit in memory (because then hash join might have been a better choice) and the columns on which join is happening are indexed.

In this case we took the large relation and we indexed both of them on the basis of the Id attribute on which join happened.

Question 8)

Using above query and seeing if the algorithm changes:

Query: select * from takes join student on takes.id = student.id order by student.id limit 10;

On using this query the algorithms/ execution plan is not changing so we have to try with another query.

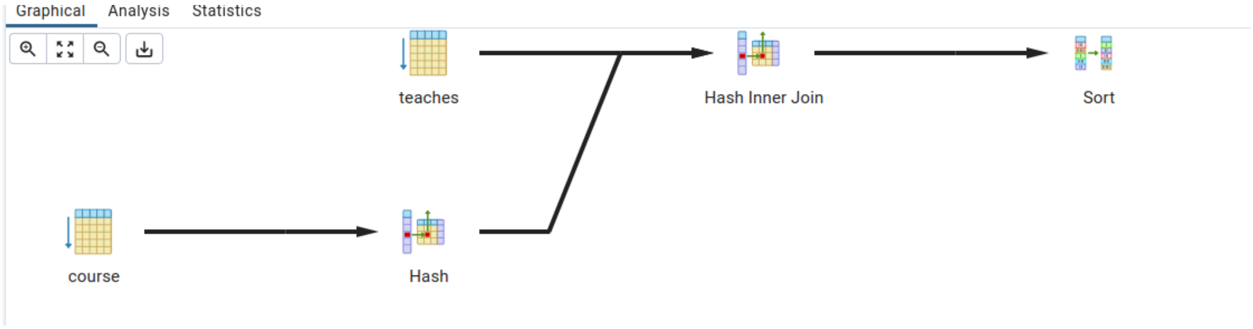
Consider the following two queries (only difference is of limit condition)

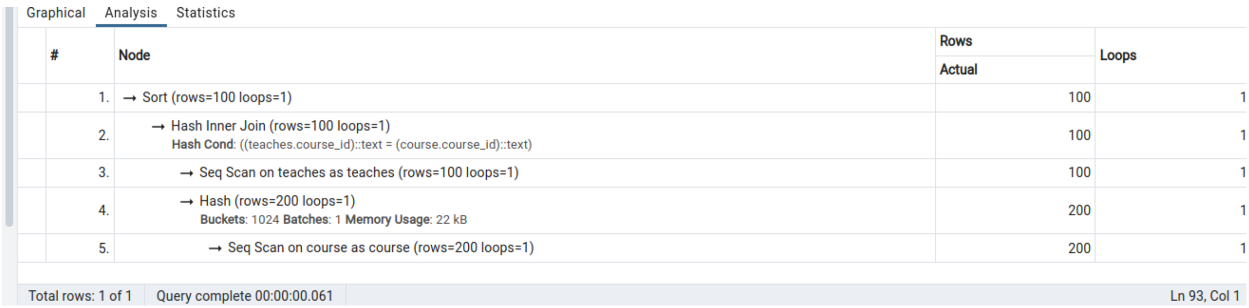
1) select * from course join teaches on teaches.course_id = course.course_id order by course.course_id ;

Result:

```
Successfully run. Total query runtime: 66 msec.
1 rows affected.
```

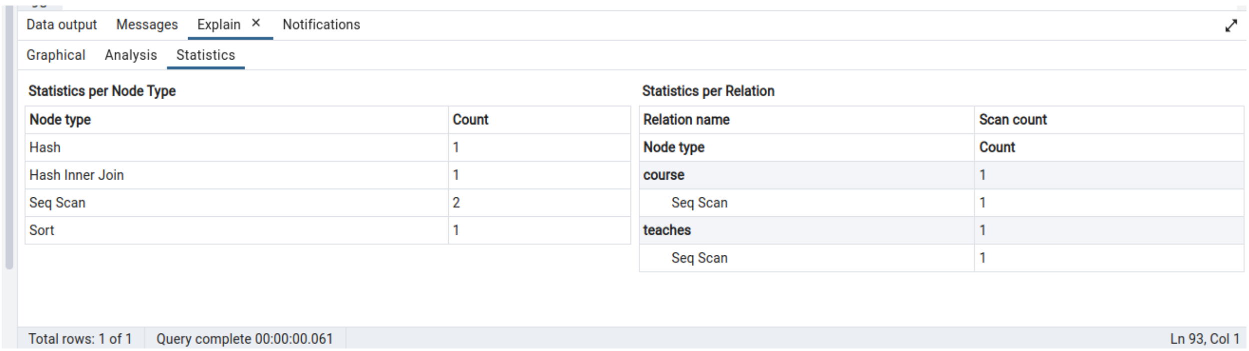
Evaluation plan:





#	Node	Rows	Loops
		Actual	
1.	→ Sort (rows=100 loops=1)	100	1
2.	→ Hash Inner Join (rows=100 loops=1) Hash Cond: ((teaches.course_id)::text = (course.course_id)::text)	100	1
3.	→ Seq Scan on teaches as teaches (rows=100 loops=1)	100	1
4.	→ Hash (rows=200 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 22 kB	200	1
5.	→ Seq Scan on course as course (rows=200 loops=1)	200	1

Total rows: 1 of 1 Query complete 00:00:00.061 Ln 93, Col 1



Statistics per Node Type

Node type	Count
Hash	1
Hash Inner Join	1
Seq Scan	2
Sort	1

Statistics per Relation

Relation name	Scan count
Node type	Count
course	1
Seq Scan	1
teaches	1
Seq Scan	1

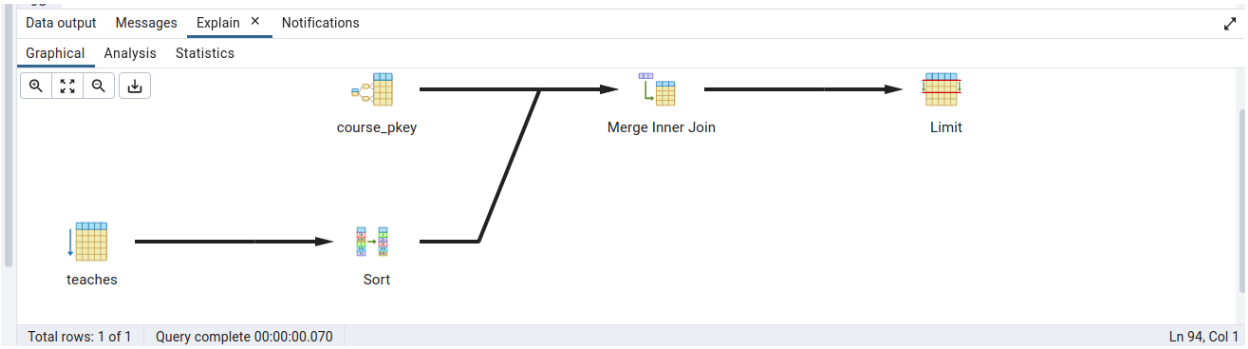
Total rows: 1 of 1 Query complete 00:00:00.061 Ln 93, Col 1

2) select * from course join teaches on teaches.course_id = course.course_id order by course.course_id limit 5;

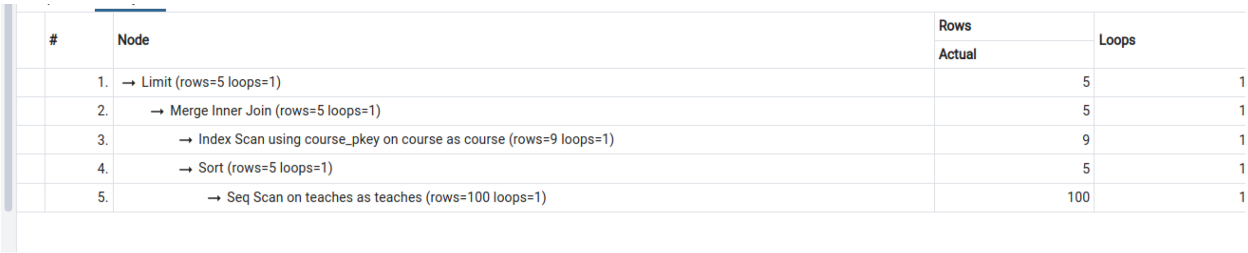
Result:

```
Successfully run. Total query runtime: 70 msec.
1 rows affected.
```

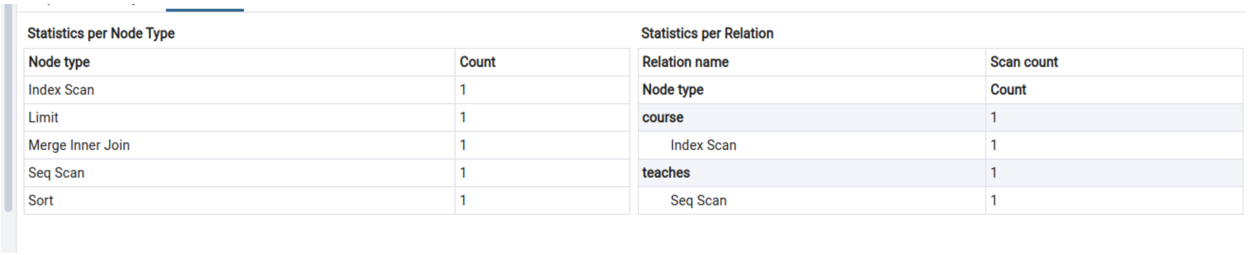
Evaluation Plan:



Total rows: 1 of 1 Query complete 00:00:00.070 Ln 94, Col 1



#	Node	Rows	Loops
		Actual	
1.	→ Limit (rows=5 loops=1)	5	1
2.	→ Merge Inner Join (rows=5 loops=1)	5	1
3.	→ Index Scan using course_pkey on course as course (rows=9 loops=1)	9	1
4.	→ Sort (rows=5 loops=1)	5	1
5.	→ Seq Scan on teaches as teaches (rows=100 loops=1)	100	1



Statistics per Node Type

Node type	Count
Index Scan	1
Limit	1
Merge Inner Join	1
Seq Scan	1
Sort	1

Statistics per Relation

Relation name	Scan count
Node type	Count
course	1
Index Scan	1
teaches	1
Seq Scan	1

The first query does following steps:

1. Sort
2. Hash Inner Join
3. Hash Cond
4. Seq Scan on teaches
5. Hash Buckets
6. Seq Scan on course

The query with limit condition does the following:

1. Limiting condition
2. Merge Inner Join
3. Index Scan using course_pkey
4. Sort
5. Seq Scan on teaches as teaches

So instead of scanning entire relations for hash joins, the limit conditions query beforehand itself, applying the limit and then doing a merge inner join followed by index scan. Saving time and space of doing entire seq scans for hashing using hash joins method.

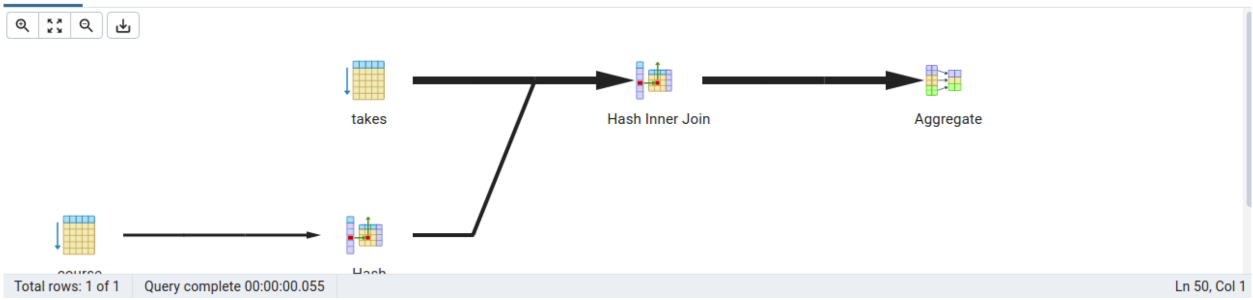
Question 9)

Aggregation query which uses hash join:

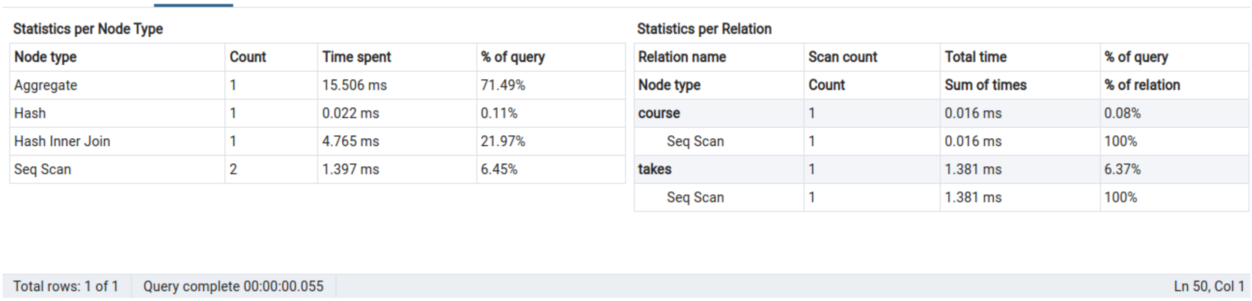
Query: select count(distinct(takes.course_id)) from takes join course on takes.course_id = course.course_id ;

Result: Successfully run. Total query runtime: 55 msec.
1 rows affected.

Execution Plan:



Total rows: 1 of 1 | Query complete 00:00:00.055 | Ln 50, Col 1



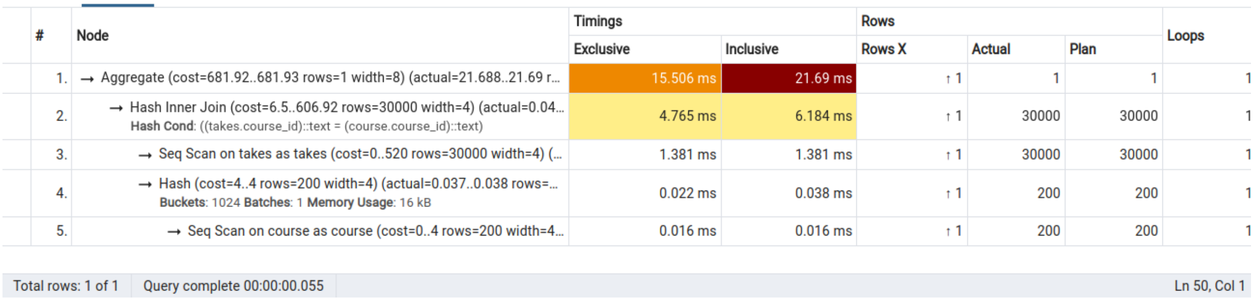
Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	1	15.506 ms	71.49%
Hash	1	0.022 ms	0.11%
Hash Inner Join	1	4.765 ms	21.97%
Seq Scan	2	1.397 ms	6.45%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
course	1	0.016 ms	0.08%
Seq Scan	1	0.016 ms	100%
takes	1	1.381 ms	6.37%
Seq Scan	1	1.381 ms	100%

Total rows: 1 of 1 | Query complete 00:00:00.055 | Ln 50, Col 1



#	Node	Timings		Rows			Loops
		Exclusive	Inclusive	Rows X	Actual	Plan	
1.	→ Aggregate (cost=681.92..681.93 rows=1 width=8) (actual=21.688..21.69 r...	15.506 ms	21.69 ms	↑ 1	1	1	1
2.	→ Hash Inner Join (cost=6.5..606.92 rows=30000 width=4) (actual=0.04... **Hash Cond**: ((takes.course_id)::text = (course.course_id)::text)	4.765 ms	6.184 ms	↑ 1	30000	30000	1
3.	→ Seq Scan on takes as takes (cost=0..520 rows=30000 width=4) (...	1.381 ms	1.381 ms	↑ 1	30000	30000	1
4.	→ Hash (cost=4..4 rows=200 width=4) (actual=0.037..0.038 rows=... **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 16 kB	0.022 ms	0.038 ms	↑ 1	200	200	1
5.	→ Seq Scan on course as course (cost=0..4 rows=200 width=4...	0.016 ms	0.016 ms	↑ 1	200	200	1

Total rows: 1 of 1 | Query complete 00:00:00.055 | Ln 50, Col 1

Since the given query is equating the one column of one table to the one column of another table. Why equality? Because Hash joins are not used in case of inequalities (!=) or range(>,<,>=,<=) queries.

Here in the execution plan we can see using a seq search on the course table, a hash is created (inner hash). Then for every tuple of takes table a hash is calculated and hash join matched resulting in an output. Now in the end the aggregation is done to get the required result

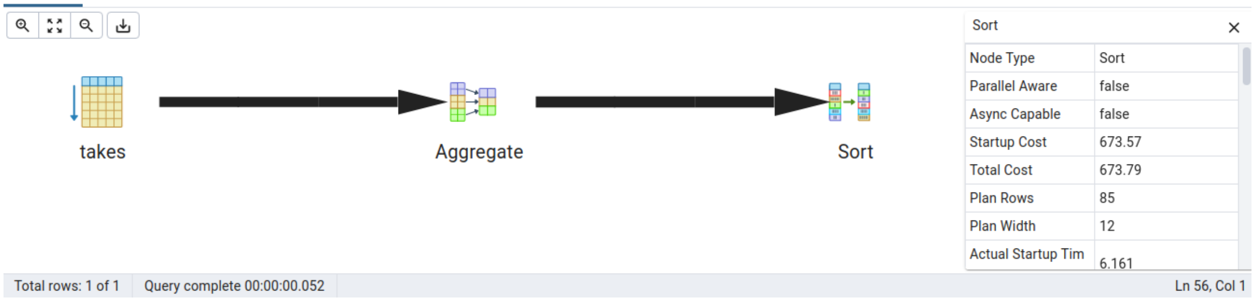
Question 10)

Aggregation query which uses sorting approach:

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

Result: Successfully run. Total query runtime: 52 msec.
1 rows affected.

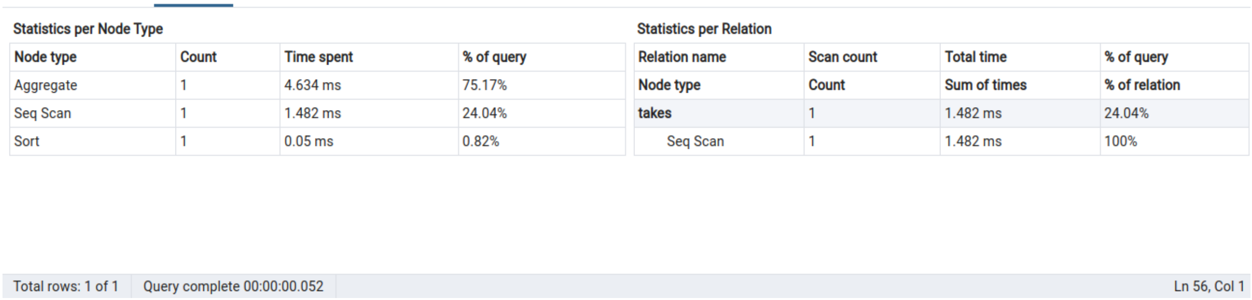
Execution Plan:



Sort

Node Type	Sort
Parallel Aware	false
Async Capable	false
Startup Cost	673.57
Total Cost	673.79
Plan Rows	85
Plan Width	12
Actual Startup Tim	6.161

Total rows: 1 of 1 Query complete 00:00:00.052 Ln 56, Col 1



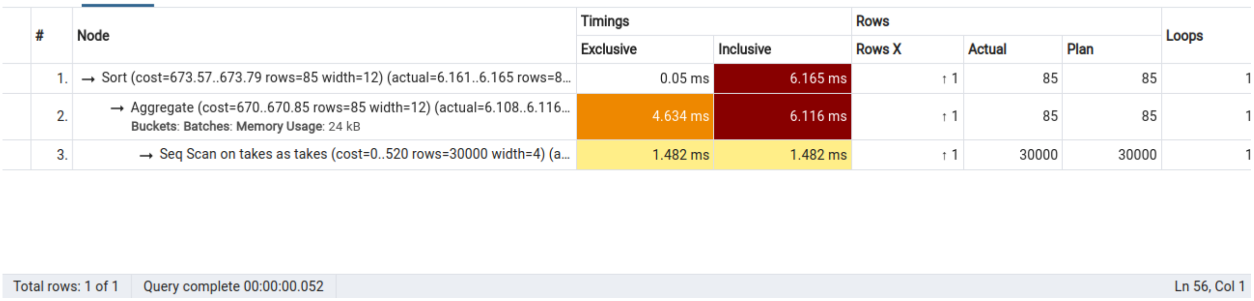
Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	1	4.634 ms	75.17%
Seq Scan	1	1.482 ms	24.04%
Sort	1	0.05 ms	0.82%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
takes	1	1.482 ms	24.04%
Seq Scan	1	1.482 ms	100%

Total rows: 1 of 1 Query complete 00:00:00.052 Ln 56, Col 1



#	Node	Timings		Rows			Loops
		Exclusive	Inclusive	Rows X	Actual	Plan	
1.	→ Sort (cost=673.57..673.79 rows=85 width=12) (actual=6.161..6.165 rows=8...	0.05 ms	6.165 ms	↑ 1	85	85	1
2.	→ Aggregate (cost=670..670.85 rows=85 width=12) (actual=6.108..6.116... Buckets: Batches: Memory Usage: 24 kB	4.634 ms	6.116 ms	↑ 1	85	85	1
3.	→ Seq Scan on takes as takes (cost=0..520 rows=30000 width=4) (a...	1.482 ms	1.482 ms	↑ 1	30000	30000	1

Total rows: 1 of 1 Query complete 00:00:00.052 Ln 56, Col 1

The aggregation function that we are using here is count which is counting the number of tuples satisfying the given condition (number of tuples groupwise). Initially all the tuples associated with the 'Gi' group are aggregated and this is done for all the distinct groups. Now we have all data grouped on the course_id attribute. Since order by clause says it to order by couse_id it has to do sorting which was our requirement

Submitted by:

Omkar Kadam 22m2112
CS 631 Course MS by Research
CSE Dept IIT Bombay 2022-23