Department of Computer Engineering

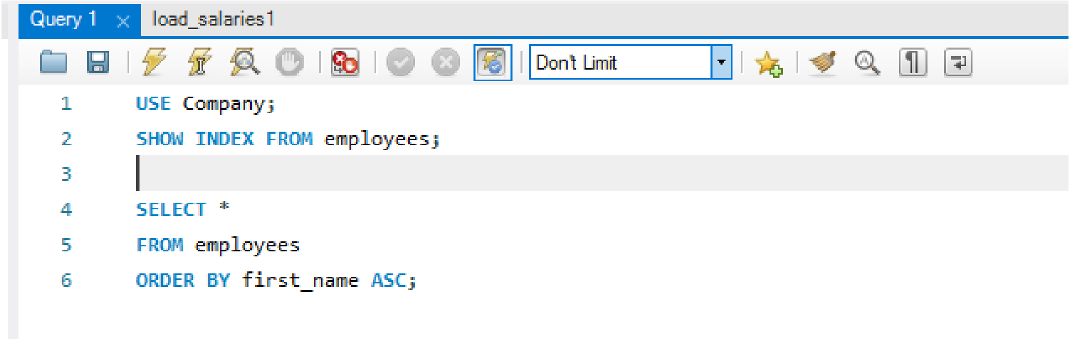
University of Peradeniya

CO527 Advanced Database Systems

Edirimanna Y.H.

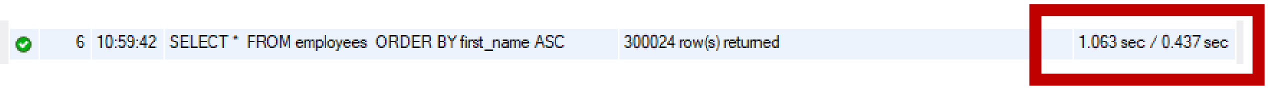
E/20/089

1. **Assuming no indexes are used, record the query execution time for retrieving all the employees by first name in ascending order.**



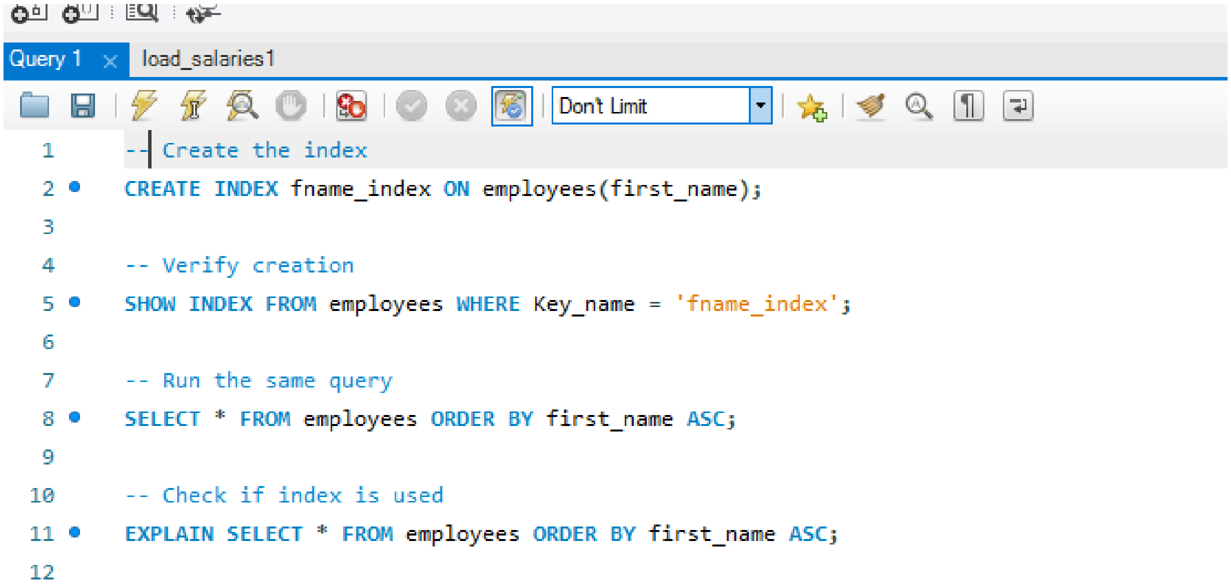
```
USE Company;
SHOW INDEX FROM employees;

SELECT *
FROM employees
ORDER BY first_name ASC;
```

time



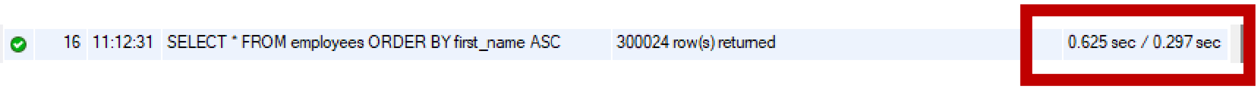
```
6  10:59:42  SELECT * FROM employees  ORDER BY first_name ASC    300024 row(s) returned    1.063 sec / 0.437 sec
```

2. **Create an index called fname_index on the first_name of the employee table. Retrieve all the employees by first name and record the query execution time. Observe the performance improvement gained when accessing with index.**



```
-- Create the index
CREATE INDEX fname_index ON employees(first_name);

-- Verify creation
SHOW INDEX FROM employees WHERE Key_name = 'fname_index';

-- Run the same query
SELECT * FROM employees ORDER BY first_name ASC;

-- Check if index is used
EXPLAIN SELECT * FROM employees ORDER BY first_name ASC;

```

Time

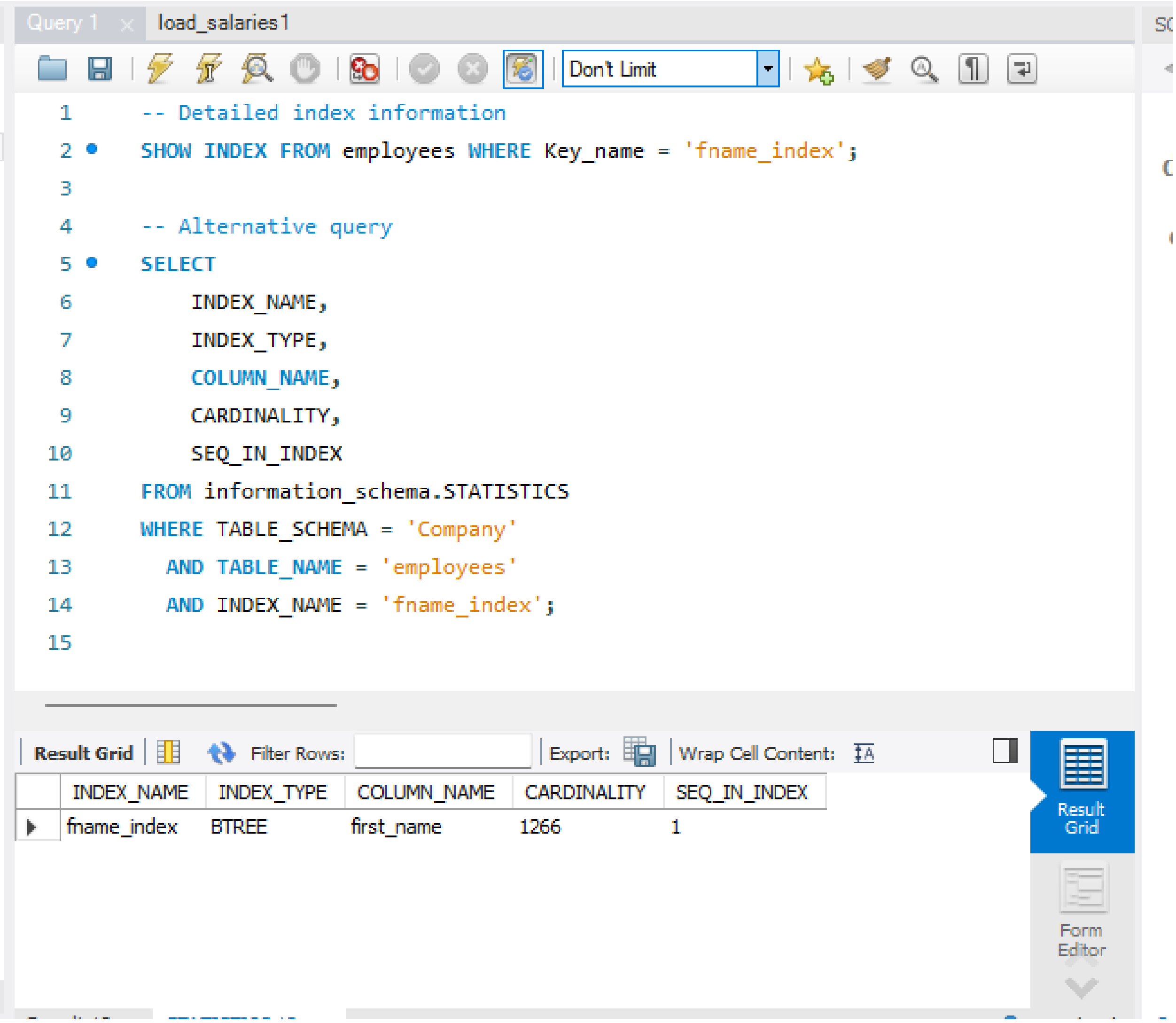


```
16  11:12:31  SELECT * FROM employees ORDER BY first_name ASC  300024 row(s) returned  0.625 sec / 0.297 sec
```

Before time is 1.063msec

After indexing time is 0.625msec

3. Which indexing technique has been used when creating the above index?

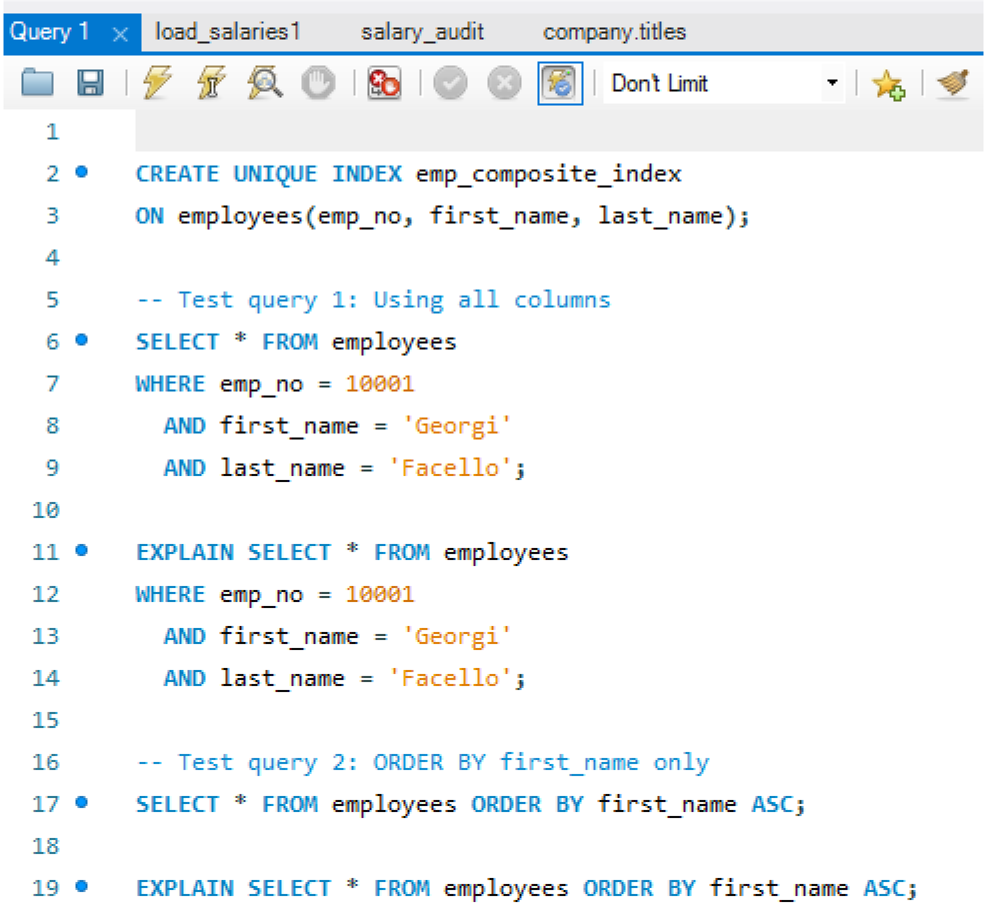


```sql
-- Detailed index information
SHOW INDEX FROM employees WHERE Key_name = 'fname_index';

-- Alternative query
SELECT
    INDEX_NAME,
    INDEX_TYPE,
    COLUMN_NAME,
    CARDINALITY,
    SEQ_IN_INDEX
FROM information_schema.STATISTICS
WHERE TABLE_SCHEMA = 'Company'
  AND TABLE_NAME = 'employees'
  AND INDEX_NAME = 'fname_index';
```

INDEX_NAME	INDEX_TYPE	COLUMN_NAME	CARDINALITY	SEQ_IN_INDEX
fname_index	BTREE	first_name	1266	1

Type is B Tree indexing

4. **Create a unique index on emp_no, first_name and last_name of employees table. Retrieve all the employees by emp_no, first_name and last_name. Observe if there is any performance improvement with respect to question1. If not, explain any possible reason.**

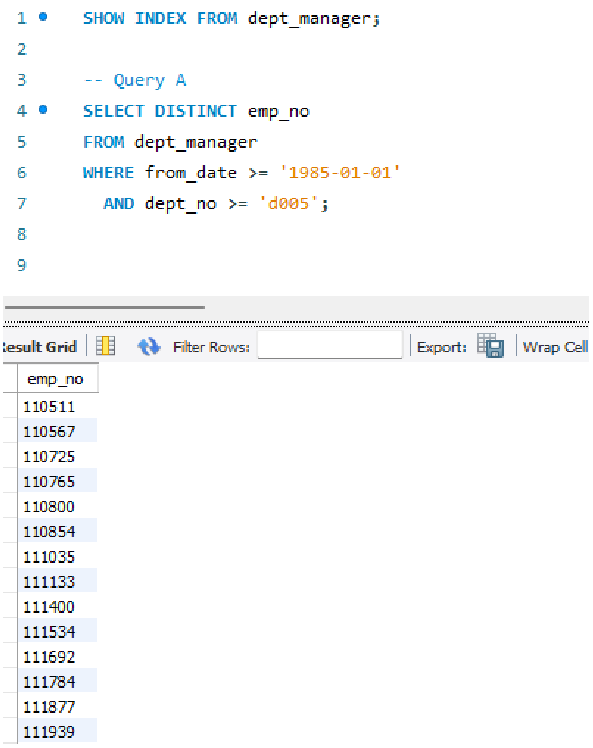


```sql

CREATE UNIQUE INDEX emp_composite_index
ON employees(emp_no, first_name, last_name);

-- Test query 1: Using all columns
SELECT * FROM employees
WHERE emp_no = 10001
  AND first_name = 'Georgi'
  AND last_name = 'Facello';

EXPLAIN SELECT * FROM employees
WHERE emp_no = 10001
  AND first_name = 'Georgi'
  AND last_name = 'Facello';

-- Test query 2: ORDER BY first_name only
SELECT * FROM employees ORDER BY first_name ASC;

EXPLAIN SELECT * FROM employees ORDER BY first_name ASC;
```

There may not be significant performance improvement compared to Question 1 because:
1. The primary key (emp_no) already has a clustered index
2. When ordering by first_name alone, the composite index (emp_no, first_name, last_name) may not be optimal since emp_no is the leading column
3. For the composite index to be effective, queries should use emp_no in WHERE clause or ORDER BY should include emp_no as the first column

5. Take the following 3 queries.

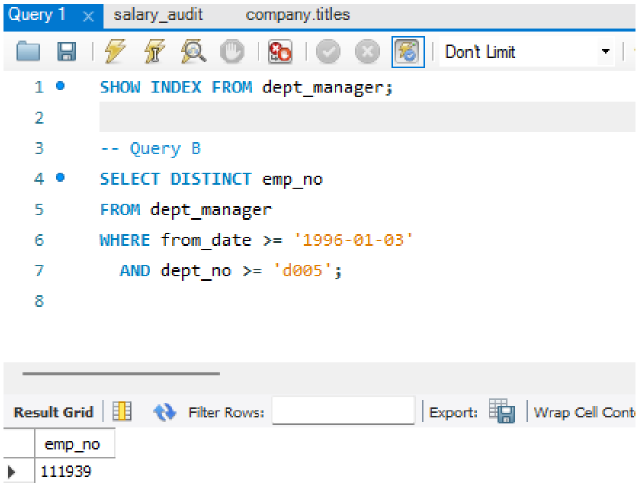
A. select distinct emp_no from dept_manager where from_date>= ’1985-01-01’ and dept_no>= ’d005’;



```
SHOW INDEX FROM dept_manager;

-- Query A
SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1985-01-01'
  AND dept_no >= 'd005';
```

```
emp_no
110511
110567
110725
110765
110800
110854
111035
111133
111400
111534
111692
111784
111877
111939
```

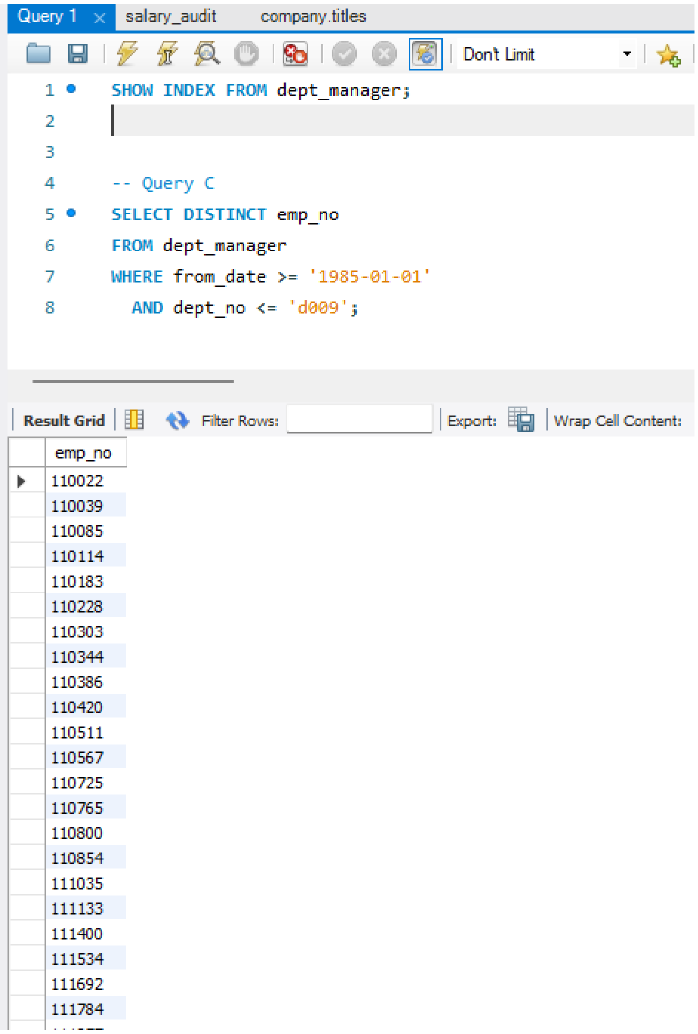
B. select distinct emp_no from dept_manager where from_date>= ’1996-01-03’ and dept_no>= ’d005’;



```
SHOW INDEX FROM dept_manager;

-- Query B
SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1996-01-03'
  AND dept_no >= 'd005';
```

```
emp_no
111939
```

C. select distinct emp_no from dept_manager where from_date>= ’1985-01-01’ and dept_no<= ’d009’;



```sql
SHOW INDEX FROM dept_manager;


-- Query C
SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1985-01-01'
  AND dept_no <= 'd009';
```

```
emp_no
110022
110039
110085
110114
110183
110228
110303
110344
110386
110420
110511
110567
110725
110765
110800
110854
111035
111133
111400
111534
111692
111784
```

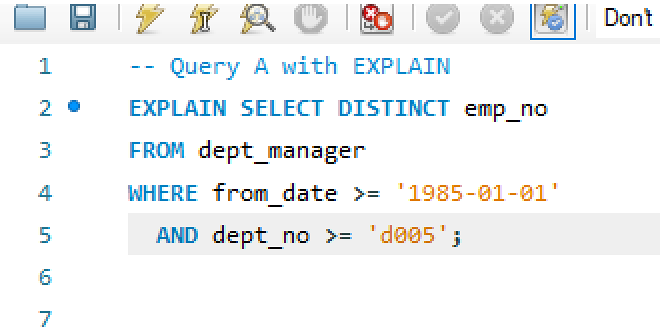
I. **Choose one single simple index(i.e index on one attribute) that is most likely to speed up all 3 queries giving reasons for your selection.**

I chose from_date because:

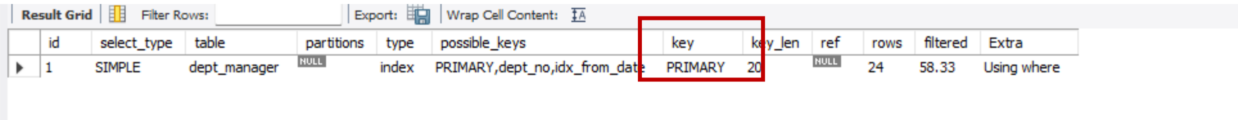
1. It appears in all three queries with range conditions, from_date has higher cardinality (more distinct values) than dept_no ,Date ranges are more selective than dept_no ranges and MySQL can use this index to filter rows before applying dept_no condition

II. For each of the 3 queries, check if MySQL storage engine used that index. If not, give a short explanation why not. You can prefix your select queries with EXPLAIN EXTENDED or with EXPLAIN to display a query execution plan.

A



```sql
-- Query A with EXPLAIN
EXPLAIN SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1985-01-01'
  AND dept_no >= 'd005';
```



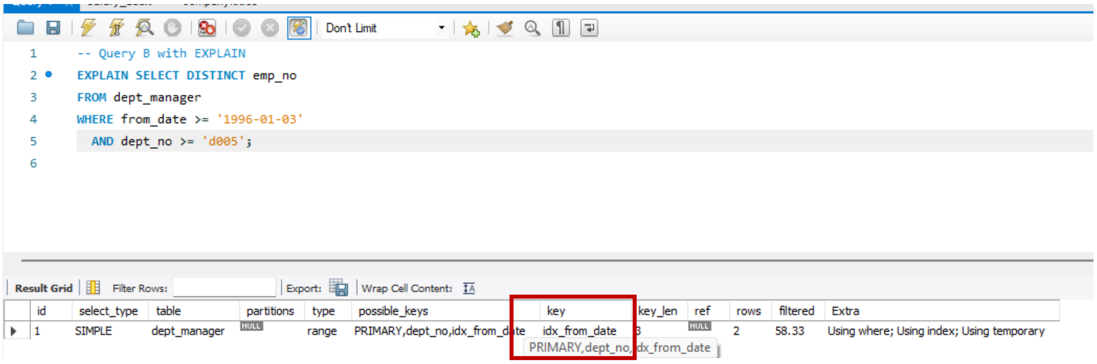
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	dept_manager	NULL	index	PRIMARY,dept_no,idx_from_date	PRIMARY	20	NULL	24	58.33	Using where

it shows "PRIMARY": Using the primary key instead.

Reason:

1. from_date = '1985-01-01' is VERY EARLY (near minimum date in table)
   - This condition matches MOST rows in the table (low selectivity)
2. dept_no >= 'd005' eliminates departments d001-d004
   - The PRIMARY KEY includes dept_no, making it useful for this filter
3. Since most rows match from_date condition, MySQL decides:
   - Cost of idx_from_date: scan many index entries + lookup rows
   - Cost of PRIMARY: direct scan with dept_no filtering built-in
   - PRIMARY is more efficient because dept_no is part of the key

B



```sql
-- Query B with EXPLAIN
EXPLAIN SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1996-01-03'
  AND dept_no >= 'd005';
```

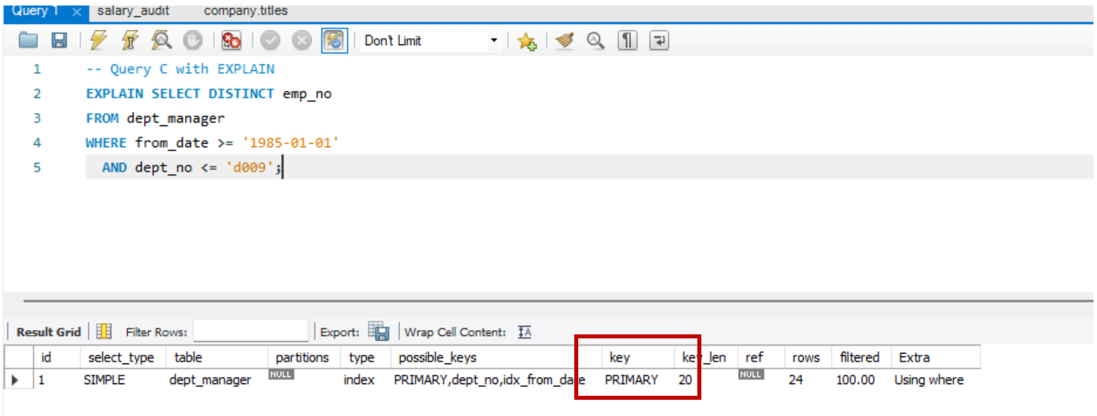
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	dept_manager	NULL	range	PRIMARY,dept_no,idx_from_date	idx_from_date	3	NULL	2	58.33	Using where; Using index; Using temporary

it shows "idx_from_date": The index is being used

1. from_date = '1996-01-03' is MUCH LATER (closer to maximum date)
   - This condition is HIGHLY SELECTIVE (matches fewer rows)
   - Most managers started before 1996, so this filters out many rows
2. The idx_from_date can quickly skip to 1996 entries

3. Since from_date filters out most rows efficiently:
   - Cost of idx_from_date: scan few index entries + lookup few rows
   - Cost of PRIMARY: must scan more entries to filter
   - idx_from_date is more efficient due to high selectivity

C



```sql
-- Query C with EXPLAIN
EXPLAIN SELECT DISTINCT emp_no
FROM dept_manager
WHERE from_date >= '1985-01-01'
  AND dept_no <= 'd009';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	dept_manager	NULL	index	PRIMARY,dept_no,idx_from_date	PRIMARY	20	NULL	24	100.00	Using where

it shows "PRIMARY": Using the primary key instead.

1. from_date = '1985-01-01' matches MOST rows (low selectivity)
2. dept_no <= 'd009' matches ALL or MOST departments
   - If there are only 9 departments (d001-d009), this matches everything
   - Very low selectivity for dept_no condition
3. PRIMARY KEY scan is efficient because:
   - Both conditions have low selectivity
   - No advantage to using idx_from_date
   - PRIMARY provides better data locality (clustered index)

6.nConsider the queries you wrote for questions 2 - 10 in PreLab assignment. Give with short explanations, which attributes on which relations should be used for creating indexes that could speed up your queries.

Query #	Query Description	Tables Involved	Recommended Index(es)	Column(s)	Reasoning
2	Top 10 family names	employees	**None**	-	Requires full table scan for GROUP BY on all last_names. Index won't help when aggregating all distinct values.
3	Number of Engineers per department	departments, dept_emp, titles	`idx_title` `idx_dept_emp_todate`	title to_date	• Filter titles containing 'Engineer'

Query #	Query Description	Tables Involved	Recommended Index(es)	Column(s)	Reasoning
4	Female managers who were senior engineers	employees, dept_manager, titles	`idx_gender` `idx_title`	gender title	• Filter current employees (to_date = '9999-01-01') • Equality filter on gender = 'F' • Equality filter on title = 'Senior Engineer'

5	Employees with salary > 115000	employees, dept_emp, departments, titles, salaries	`idx_salary`
			`idx_salaries_todate`
			`idx_dept_emp_todate`
				salary
				to_date
				to_date
					• Range query on salary > 115000
					• Filter current salaries
					• Filter current employees

6	Senior employees (>50 years, >10 years service)	employees	`idx_birth_date` `idx_hire_date`	birth_date hire_date	• Help with date range calculations • Though TIMESTAMPDIFF function prevents direct index usage, indexes still assist with range scans
7	Employees NOT in HR	employees, dept_emp, departments	`idx_dept_name` `idx_dept_emp_todate`	dept_name to_date	• Filter by department name • Filter current employees
8	Employees earning more than all Finance employees	employees, salaries, dept_emp, departments	`idx_salary` `idx_dept_emp_composite`	salary (dept_no, to_date)	• Salary comparison with MAX() • Composite index for department filtering
9	Employees earning more than company average	employees, salaries, dept_emp, departments	`idx_salary_todate`	(salary, to_date)	• Composite index for salary comparison • Filter current salaries efficiently
10	Salary difference: Senior Engineers vs All	salaries, titles	`idx_title` `idx_salaries_todate`	title to_date	• Filter by title = 'Senior Engineer' • Filter current salaries for AVG calculation

7.Assume that most of the queries on a relation are insert/update/delete. What will happen to the query execution time if that relation has an index created?

If a relation has indexes and most queries are INSERT/UPDATE/DELETE operations, the query execution time will INCREASE

- **With indexes:** Fast SELECT, Slow INSERT/UPDATE/DELETE
- **Without indexes:** Slow SELECT, Fast INSERT/UPDATE/DELETE
- Ex –
- Table with 5 indexes: - 1 INSERT operation = 1 table write + 5 index updates = 6 total operations - Without indexes = 1 table write = 1 operation Result: 6x slower for writes

For write-heavy (OLTP) systems, minimize indexes and only create essential ones. For read-heavy (OLAP/reporting) systems, more indexes are acceptable as the read performance gain outweighs the write penalty.