INFO20003 Week 8 Lab

Query Optimisation in MySQL

In this lab, you will use MySQL Workbench to see a graphical representation of the execution plan for a SQL statement. You will compare execution plans and query costs, and you will create and drop indexes to optimize query performance.

Objectives:

By the end of this lab, you should be able to:

1. Use MySQL Execution Plan to understand query execution paths
2. Understand the CREATE INDEX and DROP INDEX syntax
3. Optimize query costs by using indexes to change an execution path

Section 1: Install the Lab 8 schema

To practice query optimisation, we need to work with a much larger table than we have been using so far this semester. The *lab8v4.sql* script creates a *clients* table containing more than 50,000 rows, along with a *countries* lookup table.

◆ **Task 1.1** Open MySQL Workbench and connect to your preferred MySQL server.

◆ **Task 1.2** Download and run the *lab8v4.sql* script.

The Lab 8 script creates two new tables, *clients* and *countries*, and inserts rows into these tables.
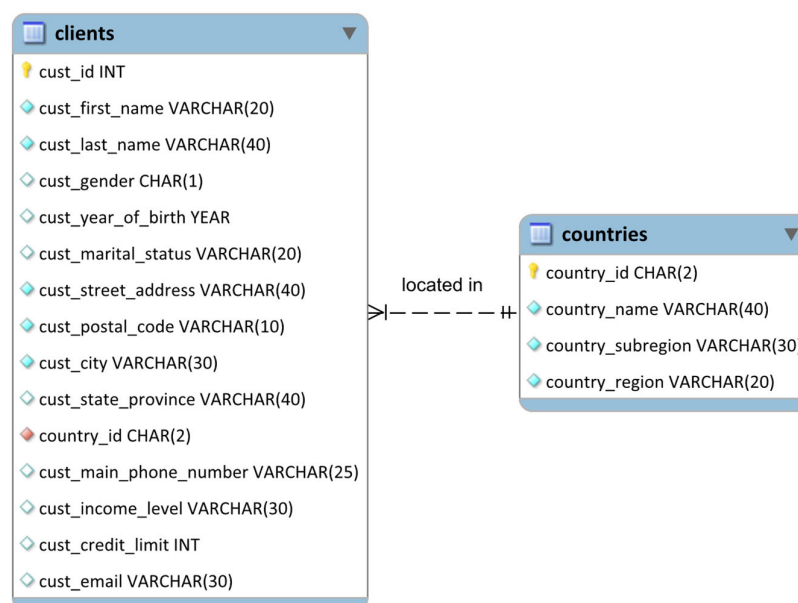

Figure 1: Lab 8 ER model

Section 2: Viewing execution plans in MySQL Workbench

The MySQL Workbench Execution Plan tool provides an explanation of how MySQL executes a query. To view an execution plan, you need to follow the steps below.

◆ **Task 2.1** Run the following query:

```sql
SELECT cust_state_province, COUNT(*)
FROM clients
WHERE country_id = 'AU'
GROUP BY cust_state_province;
```

This should return 6 rows of data:

cust_state_province	COUNT(*)
▶ Australian Capital Territory	43
New South Wales	152
Northern Territory	175
Queensland	199
South Australia	158
Victoria	104

◆ **Task 2.2** Resize the results panel so you can see the **Execution Plan** button (highlighted in red in Figure 2).
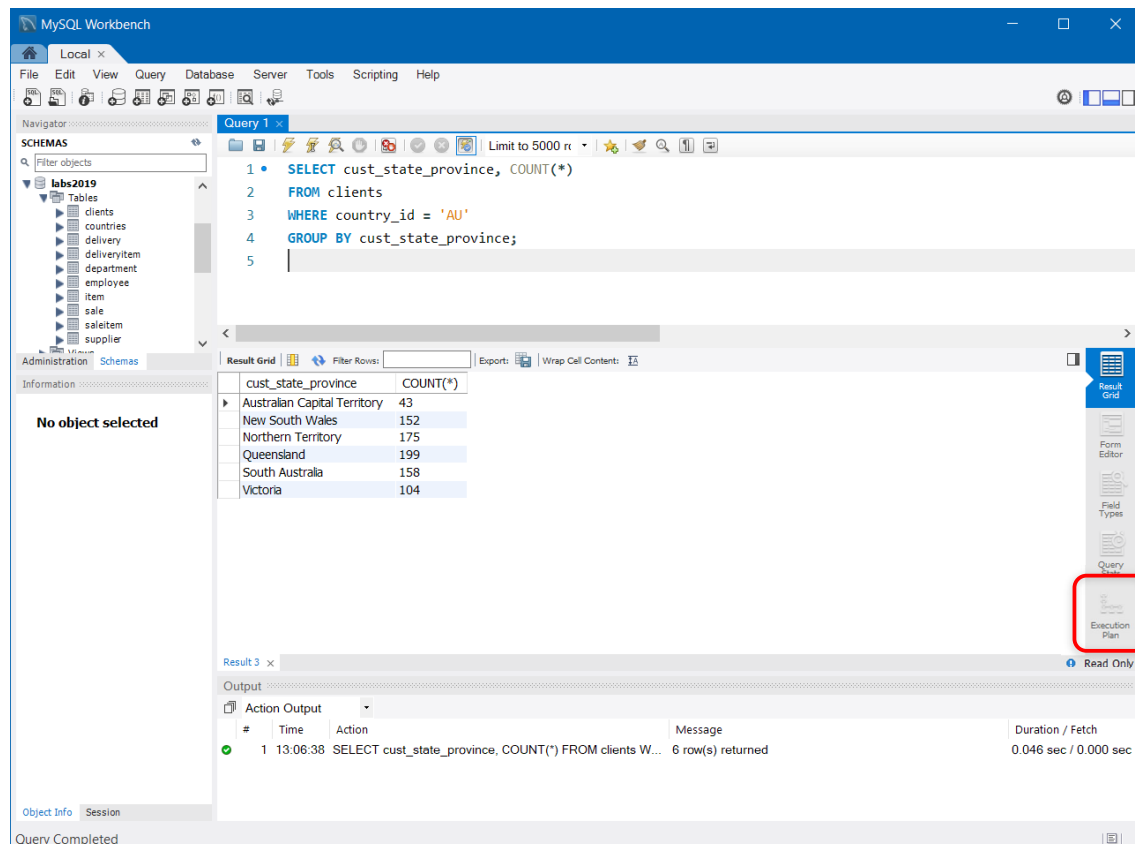

Figure 2: The Execution Plan button is shown when the results panel is tall enough.

◆ **Task 2.3** Click on **Execution Plan** to view the execution plan for this query.
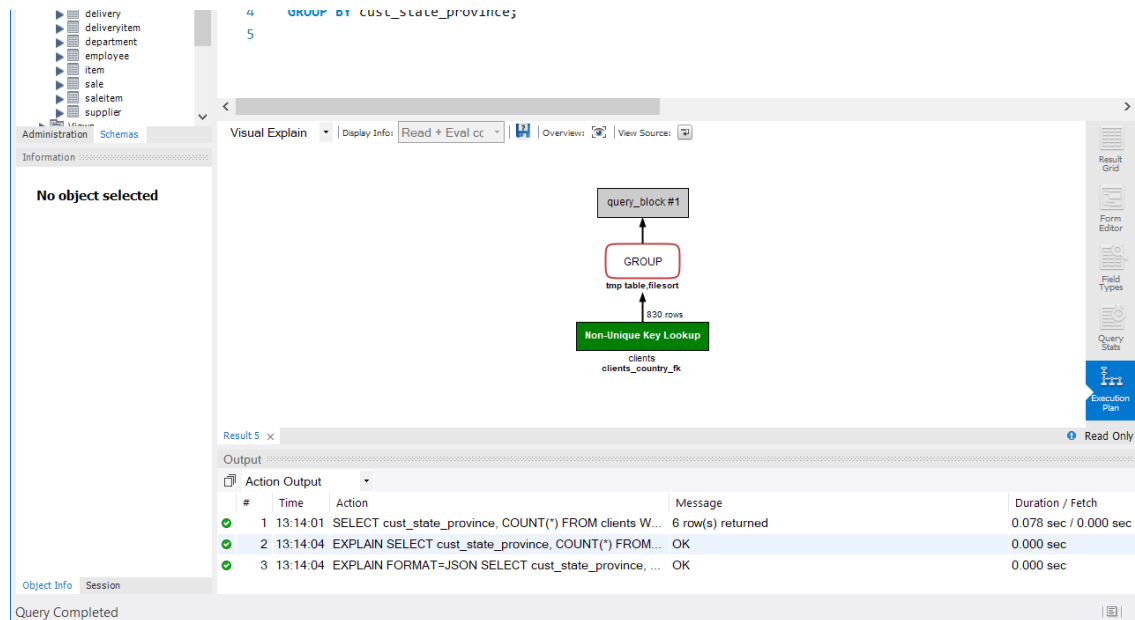

Figure 3: Workbench displays the execution plan for the query that was run.

Continued over page…

◆ **Task 2.4** Using the Execution Plan tool, investigate how the database has executed this query and what the query cost is for the SQL statement.
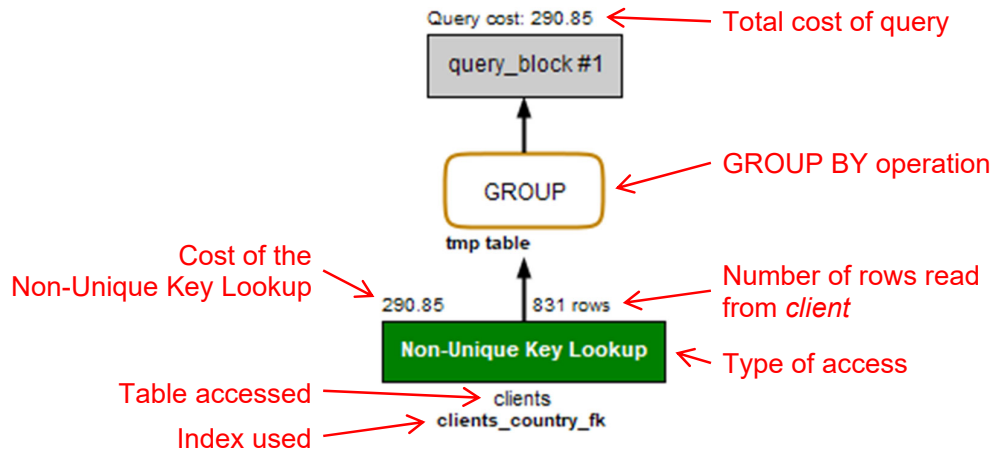

Figure 4: Execution plan for the query in Task 2.1

The execution plan in Figure 4 should be read from bottom to top. It lists the following information about execution path and cost:

- **Table accessed:** *clients* (green indicates the cost is low)

- **Type of access:** Non-Unique Key Lookup using the *clients_country_fk* index (an index on the *country_id* foreign key)

- **Cost of performing Non-Unique Key Lookup:** 290.85

- **Number of rows read from *clients* table with Non-Unique Key Lookup:** 831 rows

- **The rounded box** indicates that a GROUP BY was done

- **The total cost of executing the query:** 290.85

You can hover the mouse over a box in the plan to find out more details about it.

The lower the cost of the query, the better the performance. The use of the *clients_country_fk* index is better than a Full Table Scan (heap scan) of the clients table.

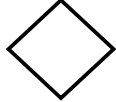> **PLEASE NOTE:** Your exact query cost may differ from those displayed in this lab. It will depend on many factors (disk characteristics, database parameters, etc). When optimising queries, it is the *change in the cost*, not the cost itself, that is an indication of query optimisation.

Interpreting the execution plan

The execution plan is a visual representation of the query cost and execution path.
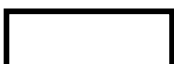
Each of the shapes and colours has a meaning in the graphical representation of the execution plan of any given query.

Table 1: Symbols used in the graphic representation of EXPLAIN plans

Graphic convention		Meaning
Standard Boxes		Tables
Rounded Boxes		Operations such as GROUP BY or sort (ORDER BY)
Framed Boxes		Subqueries
Diamonds		Joins

The colours of standard and rounded boxes also have a meaning:

Table 2: The meaning of each colour in the execution plan

Colour		Cost
Blue		Very low cost (primary key equality query)
Green		Low cost
Orange		Medium cost for index scan or temp table Low cost for unique or non-unique subquery
Red		High cost, especially for large indexes Very high cost for full table scans
Black		Default or unknown

Section 3: Measuring query performance

In this section, you will use the Execution Plan tool to obtain the execution path and cost of a set of queries. In Section 4 you will explore ways to reduce the cost of these queries.

◆ **Task 3.1** Run the following queries, and for each one, record the number of rows returned, the total cost of the query, and which indexes were used (if any).

Hint: the number of rows is found in the "Output" pane at the bottom of the Workbench window.

	Number of rows returned	Cost of query	Indexes used
Query 1			
Query 2			
Query 3			
Query 4			

Query 1:

```
SELECT cust_id, cust_last_name, cust_year_of_birth,
  cust_state_province
FROM clients
WHERE cust_state_province IN ('Victoria', 'New South Wales');
```

Query 2:

```
SELECT cust_city, cust_year_of_birth, COUNT(*)
FROM clients
WHERE cust_state_province = 'IL'
GROUP BY cust_city, cust_year_of_birth;
```

Query 3:

```
SELECT country_name, cust_state_province, COUNT(*)
FROM countries NATURAL JOIN clients
WHERE country_name = 'United Kingdom'
GROUP BY country_name, cust_state_province
ORDER BY COUNT(*) DESC;
```

Query 4:

```
SELECT country_id, cust_state_province, cust_city, COUNT(*)
FROM clients
GROUP BY country_id, cust_state_province, cust_city
ORDER BY COUNT(*) DESC;
```

Section 4: Practical query optimisation using indexes

In this section, you will view database metadata about indexes. You will create B-tree indexes to influence the execution plan of a query with the aim to improve the cost of the query.

Viewing information about existing indexes

The command

```
SHOW INDEXES FROM table_name;
```

displays all indexes on a table. For the tables we have provided (both in the department store schema and this week's schema), the indexes are only on the table constraints (primary key and foreign key).

◆ **Task 4.1** Type the following command to show the indexes on the *clients* table:

```
SHOW INDEXES FROM clients;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
clients	0	PRIMARY	1	cust_id	A	54687	NULL	NULL		BTREE
clients	0	clients_pk	1	cust_id	A	54687	NULL	NULL		BTREE
clients	1	clients_country_fk	1	country_id	A	36	NULL	NULL		BTREE

Figure 7: The indexes currently in use on the clients table

Three indexes are returned. The *clients_pk* index has *cust_id* as its search key – it is performing the role of enforcing the Primary Key as well as being a unique index. The *clients_country_fk* index has *country_id* as its search key – this index is used for the foreign key relationship from the *clients* table to the *countries* table. All three indexes are B-tree indexes, and the *Cardinality* column shows the number of distinct values in the index.

Note: the clients_countries schema we provided results in these first two rows being identical indexes; there's no need for a UNIQUE constraint on cust_id since cust_id is already the PRIMARY KEY for that table (which enforces uniqueness already!). Having these 2 duplicated indexes just means more effort when inserting a row into the customers table… so better to remove the clients_pk UNIQUE constraint from the schema if we were actually using this in a real system!

◆ **Task 4.2** Try the SHOW INDEXES command for other tables in your database. For example:

```
SHOW INDEXES FROM countries;
SHOW INDEXES FROM saleitem; -- this table is from the SQL lab schema
```

Creating indexes to influence query cost

We have studied the benefits of B-tree and Hash indexes in class. In MySQL Community Server, when using the default InnoDB storage engine, the only available type of index is the B-Tree index. We will thus be using only B-Tree indexes to improve the query performance of the five SQL statements below.

The syntax to create an index is:

```
CREATE INDEX index_name
ON table_name (column1, ...);
```

The syntax to drop the index is:

```
DROP INDEX index_name
ON table_name;
```

We will be using indexes to change the execution plan and query cost of the three SQL statements.

◆ **Task 4.3** Run the following query and view its execution plan:

```
SELECT cust_first_name, cust_last_name, cust_marital_status,
  cust_city, cust_income_level
FROM clients
WHERE cust_last_name = 'Parkburg'
  AND cust_first_name = 'Peter'
  AND cust_city = 'Trafford';
```

Query cost: 5545.35

query_block #1

5545.35   53.93K rows

Full Table Scan
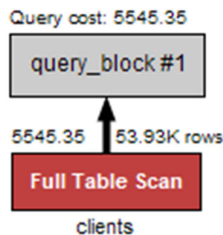
clients

You can see that this query performs a Full Table Scan of the *clients* table, reading 53,930 rows. This is because none of the three columns in the WHERE clause are indexed.

To try and improve the query performance, we could create indexes on these columns.

◆ **Task 4.4** Create an index on the *cust_city* column:

```
CREATE INDEX cust_city_idx
ON clients (cust_city);
```

Note: This command will not return any output. Have a look at the Output panel and you can see the successful execution of the command.

◆ **Task 4.5** Then rerun the query from Task 4.3 and view the execution plan again:

Query cost: 30.45

query_block #1

21.84   87 rows

Non-Unique Key Lookup

clients
cust_city_idx
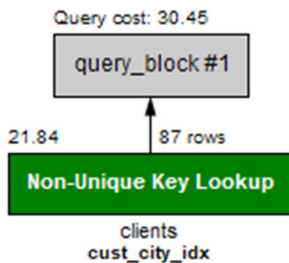
The creation of the index on the cust_city column has improved the query cost from 5545.35 to 30.45. The number of rows scanned has been reduced from 53,930 rows to 87 rows.

◆ **Task 4.6** Create another index on the *cust_last_name* and *cust_city* columns:

```
CREATE INDEX cust_last_name_city_idx
ON clients (cust_last_name, cust_city);
```

◆ **Task 4.7** Then rerun the query from Task 4.3 and view the execution plan again:
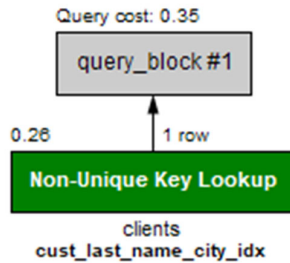
The creation of a second index on the *cust_last_name* and *cust_city* columns has resulted in only one row being returned from the Key Lookup of this index, and the I/O cost has been reduced to below 1.

◆ **Task 4.8** Drop the indexes on the *clients* table.

```
DROP INDEX cust_city_idx ON clients;
DROP INDEX cust_last_name_city_idx ON clients;
```

Your turn

Now it's your turn to choose and create appropriate indexes to speed up query execution.

◆ **Task 4.9** Look back at the first two queries from Task 3.1. Using what you have learned in this section, create one or more indexes that will improve the execution cost and plan of Queries 1 and 2. Then run the queries and fill in the following table:

	Original cost of query (Task 3.1)	**New cost of query**	**Indexes used**
Query 1			
Query 2			

To improve the cost of Query 3, you could try creating a unique index. This enforces that every value in the column is unique, which can improve the performance of equality queries even more than a non-unique index.

```
CREATE UNIQUE INDEX country_name_idx
ON countries (country_name);
```

◆ **Task 4.10** How much does the unique index improve the cost of Query 3 from Task 3.1? Fill in the table:

	Original cost of query (Task 3.1)	**New cost of query**	**Indexes used**

Query 3			

◆ **Task 4.11** Can the performance of Query 4 be improved any further? If so, how? If not, why not?

End of Week 8 Lab