

Part 1:

DDL Commands:

Country:

```
CREATE TABLE Country(  
    CountryName VARCHAR(100) PRIMARY KEY,  
    IncomeGroup VARCHAR(255),  
    CountryCode VARCHAR(255),  
    Region VARCHAR(255)  
);
```

User:

```
CREATE TABLE User (  
    UserID INT PRIMARY KEY,  
    Username VARCHAR(255),  
    Email VARCHAR(255) UNIQUE,  
    Role VARCHAR(50)  
);
```

NaturalDisaster:

```
CREATE TABLE NaturalDisaster (  
    DisasterID INT PRIMARY KEY,  
    CountryName VARCHAR(100),  
    Type VARCHAR(100),  
    Year INT,  
    Intensity FLOAT  
);
```

DirectDamage:

```
CREATE TABLE DirectDamage (  
    DamageID INT PRIMARY KEY,
```

```
DisasterID INT,  
CountryName VARCHAR(100),  
TotalDamage INT,  
TotalDamageScale INT,  
HousesDestroyed INT,  
HousesDestroyedScale INT,  
Injuries INT,  
Deaths INT,  
FOREIGN KEY (DisasterID) REFERENCES NaturalDisaster(DisasterID)  
);
```

SectoralEconomicImpact:

```
CREATE TABLE SectoralEconomicImpact (  
    CountryName VARCHAR(100),  
    Year INT,  
    AgricultureAnnualPercentGrowth FLOAT,  
    IndustryAnnualPercentGrowth FLOAT,  
    ManufacturingAnnualPercentGrowth FLOAT,  
    ServiceAnnualPercentGrowth FLOAT,  
    PRIMARY KEY (CountryName, Year)  
);
```

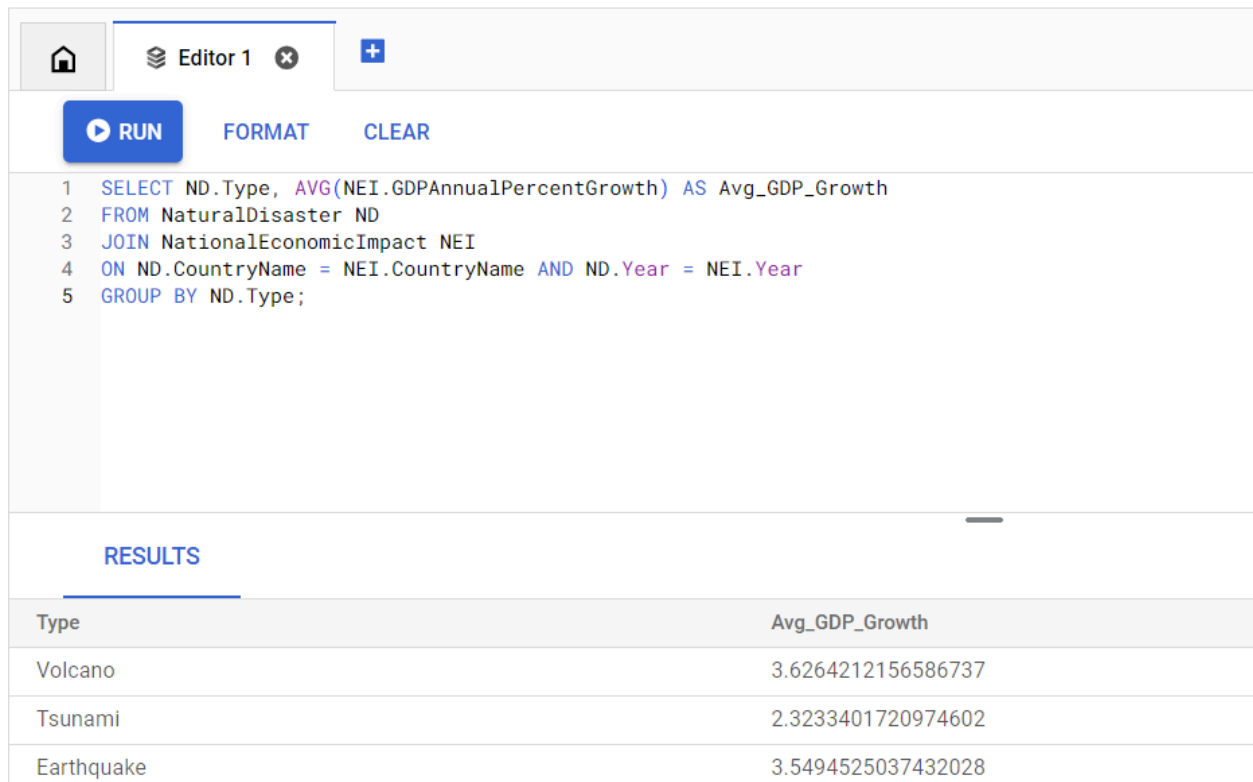
NationalEconomicImpact:

```
CREATE TABLE IF NOT EXISTS NationalEconomicImpact (  
    CountryName VARCHAR(100),  
    Year INT,  
    CPI_2010_100 FLOAT,  
    ExportsAnnualPercentGrowth FLOAT,  
    GDPAnnualPercentGrowth FLOAT,  
    ImportAnnualPercentGrowth FLOAT,  
    UnemploymentPercent FLOAT,  
    PRIMARY KEY (CountryName, Year)
```

);

Advanced Queries Results Screenshots (Examination of the queries below)

Query 1:



The screenshot shows a SQL query editor interface. At the top, there is a toolbar with a home icon, a tab labeled 'Editor 1', and a plus icon. Below the toolbar are three buttons: 'RUN' (with a play icon), 'FORMAT', and 'CLEAR'. The query text is as follows:

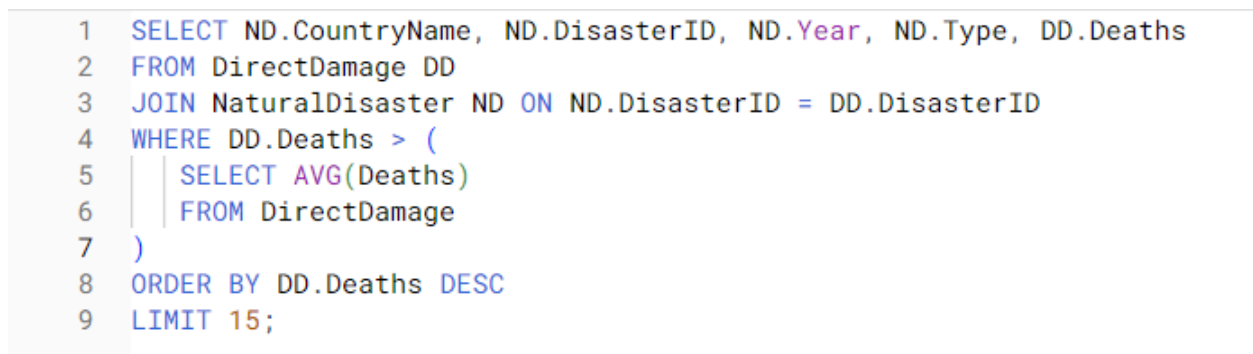
```
1 SELECT ND.Type, AVG(NEI.GDPAnnualPercentGrowth) AS Avg_GDP_Growth
2 FROM NaturalDisaster ND
3 JOIN NationalEconomicImpact NEI
4 ON ND.CountryName = NEI.CountryName AND ND.Year = NEI.Year
5 GROUP BY ND.Type;
```

Below the query editor, the 'RESULTS' tab is selected, showing a table with the following data:

Type	Avg_GDP_Growth
Volcano	3.6264212156586737
Tsunami	2.3233401720974602
Earthquake	3.5494525037432028

Note: The query implicitly can only have 3 lines of output because we are grouping by natural disaster and using only 3 natural disasters: volcanoes, tsunamis, and earthquakes.

Query 2:



The screenshot shows a SQL query editor with the following query text:

```
1 SELECT ND.CountryName, ND.DisasterID, ND.Year, ND.Type, DD.Deaths
2 FROM DirectDamage DD
3 JOIN NaturalDisaster ND ON ND.DisasterID = DD.DisasterID
4 WHERE DD.Deaths > (
5     SELECT AVG(Deaths)
6     FROM DirectDamage
7 )
8 ORDER BY DD.Deaths DESC
9 LIMIT 15;
```

RESULTS

CountryName	DisasterID	Year	Type	Deaths
China	4526	1556	Earthquake	830000
Haiti	9547	2010	Earthquake	316000
Haiti	3556	2010	Tsunami	316000
Turkmenistan	3905	115	Earthquake	260000
Turkmenistan	922	115	Tsunami	260000
Turkmenistan	3980	525	Earthquake	250000
China	8177	1976	Earthquake	242769
Azerbaijan	4190	1139	Earthquake	230000
Indonesia	9210	2004	Earthquake	227899
Indonesia	3475	2004	Tsunami	227899
China	6807	1920	Earthquake	200000
Iran, Islamic Rep.	4067	856	Earthquake	200000
Armenia	4089	893	Earthquake	150000
China, Arab Rep.	8888	588	Earthquake	100000

Rows per page: 20 1 - 15 of 15 < > >>

Query 3:

```
1 SELECT ND.Type, ND.Intensity, ROUND(AVG(DD.Deaths), 0) AS Avg_Deaths
2 FROM NaturalDisaster ND
3 JOIN DirectDamage DD ON ND.DisasterID = DD.DisasterID
4 GROUP BY ND.Intensity, ND.Type
5 ORDER BY Avg_Deaths DESC
6 LIMIT 15;
```

RESULTS

Type	Intensity	Avg_Deaths
Earthquake	9.1	82111
Tsunami	3.8	27122
Earthquake	8	15629
Volcano	7	15000
Earthquake	8.3	7882
Earthquake	8.5	6688
Tsunami	5	6245
Earthquake	7.9	5202
Earthquake	9	5000
Tsunami	4	3793
Earthquake	7	2965
Earthquake	7.5	2896
Tsunami	2.7	2500

Query 4:

```
1 SELECT
2     ND.Year, ND.CountryName,
3     COUNT(*) AS NumDisasters,
4     SEI.AgricultureAnnualPercentGrowth
5 FROM NaturalDisaster ND
6 JOIN SectoralEconomicImpact SEI
7     ON ND.CountryName = SEI.CountryName AND ND.Year = SEI.Year
8 WHERE ND.CountryName LIKE 'Indonesia'
9 GROUP BY ND.Year, SEI.AgricultureAnnualPercentGrowth, ND.CountryName
10 ORDER BY NumDisasters DESC
11 LIMIT 15;
```

Year	CountryName	NumDisasters	AgricultureAnnualPercentGrowth
2018	Indonesia	20	3.88416
2004	Indonesia	17	2.81891
1994	Indonesia	17	0.555938
2009	Indonesia	13	3.95782
2006	Indonesia	13	3.35633
2019	Indonesia	12	3.6065
1979	Indonesia	12	3.85352
2007	Indonesia	11	3.47043
2010	Indonesia	11	3.00567
2016	Indonesia	10	3.37273
1963	Indonesia	9	-3.71209
2017	Indonesia	9	3.91593
2013	Indonesia	9	4.20429
.....	-

Part 2

Advanced Queries and Indexes

QUERY 1: Calculate the average GDP growth by the type of disaster

```
SELECT ND.Type, AVG(NEI.GDPAnnualPercentGrowth) AS Avg_GDP_Growth
```

FROM NaturalDisaster ND

JOIN NationalEconomicImpact NEI

ON ND.CountryName = NEI.CountryName AND ND.Year = NEI.Year

GROUP BY ND.Type;

Before Index:

```

1 EXPLAIN
2 SELECT ND.Type, AVG(NEI.GDPAnnualPercentGrowth) AS Avg_GDP_Growth
3 FROM NaturalDisaster ND
4 JOIN NationalEconomicImpact NEI
5   ON ND.CountryName = NEI.CountryName AND ND.Year = NEI.Year
6 GROUP BY ND.Type;

```

The NaturalDisaster table is doing a full table scan, which isn't ideal. The NEI table uses its primary key, which is good (eq_ref join on (CountryName, Year)). Therefore, we want to do the scan on ND.

```
Index: CREATE INDEX idx_nd_country_year ON NaturalDisaster(CountryName, Year);
```

After Index:

```

1 EXPLAIN
2 SELECT ND.Type, AVG(NEI.GDPAnnualPercentGrowth) AS Avg_GDP_Growth
3 FROM NaturalDisaster ND
4 JOIN NationalEconomicImpact NEI
5 | ON ND.CountryName = NEI.CountryName AND ND.Year = NEI.Year
6 GROUP BY ND.Type;

```

Now, NaturalDisaster is using the new index with a ref join, which is better. NEI switched to a full scan as its rows are fewer, and it's driving the join.

Rationale:

Index: `CREATE INDEX idx_nd_country_year ON NaturalDisaster(CountryName, Year);`

For the first query, which calculates the average GDP growth by the type of disaster, the initial EXPLAIN showed that the NaturalDisaster table was performing a full table scan, searching through 9302 rows whilst it joined with the NationalEconomicImpact (NEI) table. Pre-indexing, the primary key on the NEI table, was the only one being utilised. After adding the index on the CountryName and Year columns in the NaturalDisaster table, the join accessed only one row from NaturalDisaster per match, reducing the number of rows that would be scanned. After this, however, the 4369 rows of the NEI table now get scanned instead. Regardless, the performance has improved as the new index optimises the indexed query.

QUERY 2: Get disasters with above-average deaths

```
SELECT ND.CountryName, ND.DisasterID, ND.Year, ND.Type, DD.Deaths
FROM DirectDamage DD
JOIN NaturalDisaster ND ON ND.DisasterID = DD.DisasterID
WHERE DD.Deaths > (
    SELECT AVG(Deaths)
    FROM DirectDamage
)
ORDER BY DD.Deaths DESC;
```

Before Index:

```
1 EXPLAIN
2 SELECT ND.CountryName, ND.DisasterID, ND.Year, ND.Type, DD.Deaths
3 FROM DirectDamage DD
4 JOIN NaturalDisaster ND ON ND.DisasterID = DD.DisasterID
5 WHERE DD.Deaths > (
6     SELECT AVG(Deaths)
7     FROM DirectDamage
8 )
9 ORDER BY DD.Deaths DESC;
```

RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	DD		ALL	idx_disasterid_direct				9228	33.33	Using where; Using filesort
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		ALL					9228	100.00	

Index: `CREATE INDEX idx_deaths ON DirectDamage(Deaths);`

After Index:

1	EXPLAIN
2	SELECT ND.CountryName, ND.DisasterID, ND.Year, ND.Type, DD.Deaths
3	FROM DirectDamage DD
4	JOIN NaturalDisaster ND ON ND.DisasterID = DD.DisasterID
5	WHERE DD.Deaths > (
6	SELECT AVG(Deaths)
7	FROM DirectDamage
8)
9	ORDER BY DD.Deaths DESC;

RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	DD		ALL	idx_disasterid_direct				9228	33.33	Using where; Using filesort
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoosse.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		ALL					9228	100.00	

Rationale:

Index: `CREATE INDEX idx_deaths ON DirectDamage(Deaths);`

For the second query, which gets disasters with above-average deaths, the initial EXPLAIN showed that the query was using a range scan on the DirectDamage table with 513 rows being examined. Joining with the NaturalDisaster table used the primary key, accessing a row per match. After the index on the Deaths column, the same thing was executed; the index is optimal for the query. As there was no change in query performance pre and post-indexing, we see that MySQL was already efficiently optimising the query— adding this index confirmed this and did not improve the performance.

QUERY 3: Calculate the average deaths by disaster type and intensity of the respective disaster.

Average deaths for type and intensity (join and group by)

`SELECT ND.Type, ND.Intensity, ROUND(AVG(DD.Deaths), 0) AS Avg_Deaths`

`FROM NaturalDisaster ND`

`JOIN DirectDamage DD ON ND.DisasterID = DD.DisasterID`

`GROUP BY ND.Intensity, ND.Type`

`ORDER BY Avg_Deaths DESC;`

Before Index:

1 EXPLAIN

2 SELECT ND.Type, ND.Intensity, ROUND(AVG(DD.Deaths), 0) AS Avg_Deaths

3 FROM NaturalDisaster ND

4 JOIN DirectDamage DD ON ND.DisasterID = DD.DisasterID

5 GROUP BY ND.Intensity, ND.Type

6 ORDER BY Avg_Deaths DESC;

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	DD		ALL	idx_disasterid_direct				9228	100.00	Using where; Using temporary; Using filesort
1	SIMPLE	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	

Index 1: CREATE INDEX idx_dd_disasterid_deaths ON DirectDamage(DisasterID, Deaths);

After Index:

1 EXPLAIN

2 SELECT ND.Type, ND.Intensity, ROUND(AVG(DD.Deaths), 0) AS Avg_Deaths

3 FROM NaturalDisaster ND

4 JOIN DirectDamage DD ON ND.DisasterID = DD.DisasterID

5 GROUP BY ND.Intensity, ND.Type

6 ORDER BY Avg_Deaths DESC;

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	
1	SIMPLE	DD		index	idx_disasterid_direct,idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		9228	100.00	✓
1	SIMPLE	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	

Index 2: CREATE INDEX idx_nd_disaster_type_intensity ON NaturalDisaster(DisasterID, Type, Intensity);

After Index 2:

1 EXPLAIN

2 SELECT ND.Type, ND.Intensity, ROUND(AVG(DD.Deaths), 0) AS Avg_Deaths

3 FROM NaturalDisaster ND

4 JOIN DirectDamage DD ON ND.DisasterID = DD.DisasterID

5 GROUP BY ND.Intensity, ND.Type

6 ORDER BY Avg_Deaths DESC;

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	
1	SIMPLE	DD		index	idx_disasterid_direct,idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		9228	100.00	✓
1	SIMPLE	ND		eq_ref	PRIMARY,idx_nd_disaster_type_intensity	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	

Rationale:**Index:**

1. `CREATE INDEX idx_dd_disasterid_deaths ON DirectDamage(DisasterID, Deaths);`
2. `CREATE INDEX idx_nd_disaster_type_intensity ON NaturalDisaster(DisasterID, Type, Intensity);`

The third query calculates the average deaths by disaster type and intensity of the respective disaster. The initial EXPLAIN showed how the DirectDamage table was performing a full table scan with the 9228 rows. After the first index, the one on DisasterID and Deaths in the DirectDamage table, the query began using that index, but the rows scanned remained the same. That said, the query started using the new index, not scanning without one. The join with the NaturalDisaster table continues with the primary key, getting one row a match. Even with a second index on the NaturalDisaster table, the optimiser chose the primary key again. The number of rows to scan did not change. The Extra column of the EXPLAIN table says the following: “Using where; Using index; Using temporary; Using filesort.” This means that the new index was being used as it could improve efficiency. That said, with the grouping and ordering, the “temporary” tables and filesort were still needed, so the performance didn’t improve all that much.

QUERY 4: Counts the disasters and gets the agricultural growth for Indonesia

`SELECT`

`ND.Year, ND.CountryName,`

`COUNT(*) AS NumDisasters,`

`SEI.AgricultureAnnualPercentGrowth`

`FROM NaturalDisaster ND`

`JOIN SectoralEconomicImpact SEI`

`ON ND.CountryName = SEI.CountryName AND ND.Year = SEI.Year`

`WHERE ND.CountryName LIKE 'Indonesia'`

`GROUP BY ND.Year, SEI.AgricultureAnnualPercentGrowth, ND.CountryName`

`ORDER BY NumDisasters DESC;`

Before Index:

<pre>1 EXPLAIN 2 SELECT ND.Year, ND.CountryName, 3 COUNT(*) AS NumDisasters, 4 SEI.AgricultureAnnualPercentGrowth 5 FROM NaturalDisaster ND 6 JOIN SectoralEconomicImpact SEI 7 ON ND.CountryName = SEI.CountryName AND ND.Year = SEI.Year 8 WHERE ND.CountryName LIKE 'Indonesia' 9 GROUP BY ND.Year, SEI.AgricultureAnnualPercentGrowth, ND.CountryName 10 ORDER BY NumDisasters DESC;</pre>											
RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		range	idx_nd_country_year	idx_nd_country_year	403		875	100.00	Using where; U...
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databoose.ND.CountryName,test_databoose.ND.Year	1	100.00	

Index: **CREATE INDEX** idx_nd_country_year **ON** NaturalDisaster(CountryName, Year);

THIS IS THE SAME INDEX AS IS IN *QUERY 1*.

After Index: Same as pre-index.

Rationale:

Index: **CREATE INDEX** idx_nd_country_year **ON** NaturalDisaster(CountryName, Year);

Note: No new index was added.

For the fourth query, one that counts the disasters and gets the agricultural growth for Indonesia, the EXPLAIN output highlighted that the query was already using an index that we previously created (for the first query). These indexes are the same as they both used the CountryName and Year columns in the join condition. The primary key here, (CountryName, Year), was used to join with the SectoralEconomicImpact table.

If we drop the index created, we see the following:

<pre>1 EXPLAIN 2 SELECT ND.Year, ND.CountryName, 3 COUNT(*) AS NumDisasters, 4 SEI.AgricultureAnnualPercentGrowth 5 FROM NaturalDisaster ND 6 JOIN SectoralEconomicImpact SEI 7 ON ND.CountryName = SEI.CountryName AND ND.Year = SEI.Year 8 WHERE ND.CountryName LIKE 'Indonesia' 9 GROUP BY ND.Year, SEI.AgricultureAnnualPercentGrowth, ND.CountryName 10 ORDER BY NumDisasters DESC;</pre>											
RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		ALL					9302	11.11	Using where; U...
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databoose.ND.CountryName,test_databoose.ND.Year	1	100.00	

This shows that it would have done a full-table scan; the index helped efficiency by dropping the rows scanned from 9302 to 875. Only 11.11% of the data was regarding Indonesia, making it super inefficient to scan the whole table. This shows that the index was very helpful in increasing the efficiency of the query.