

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,

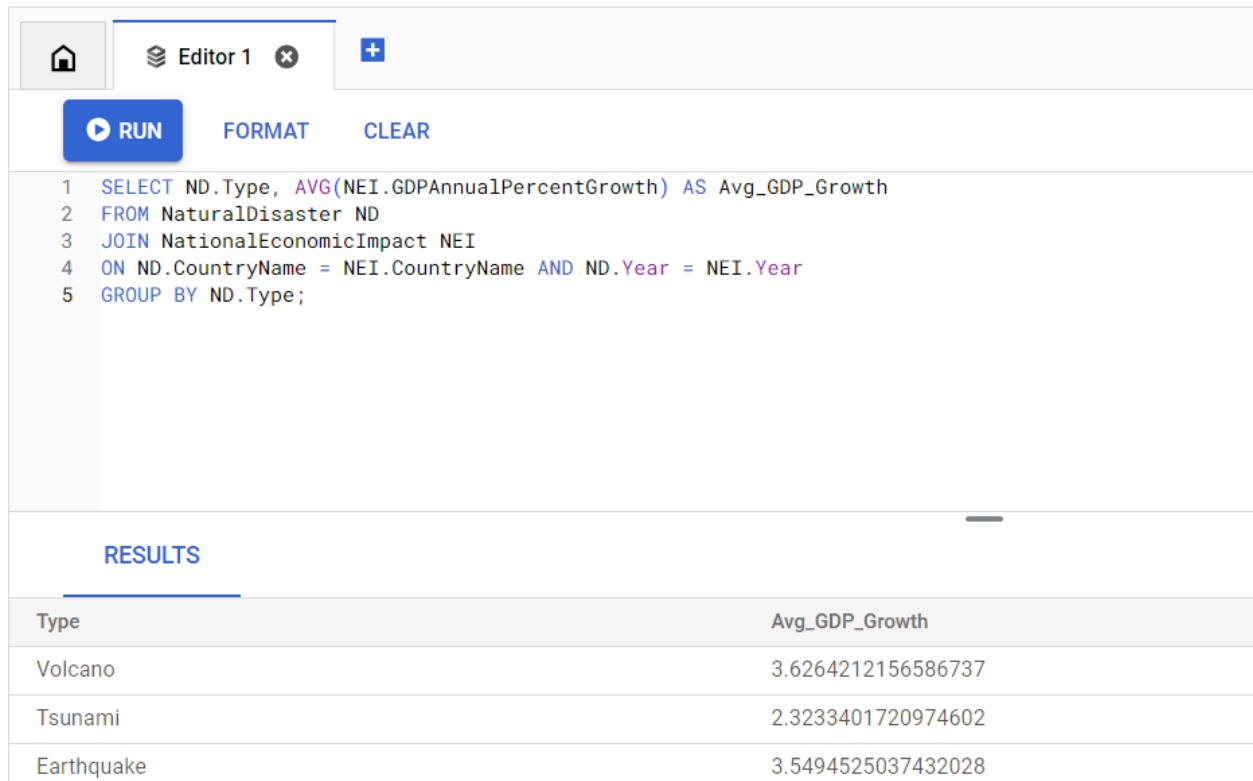
```

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, there are three buttons: 'RUN' (with a play icon), 'FORMAT', and 'CLEAR'. The query text is as follows:

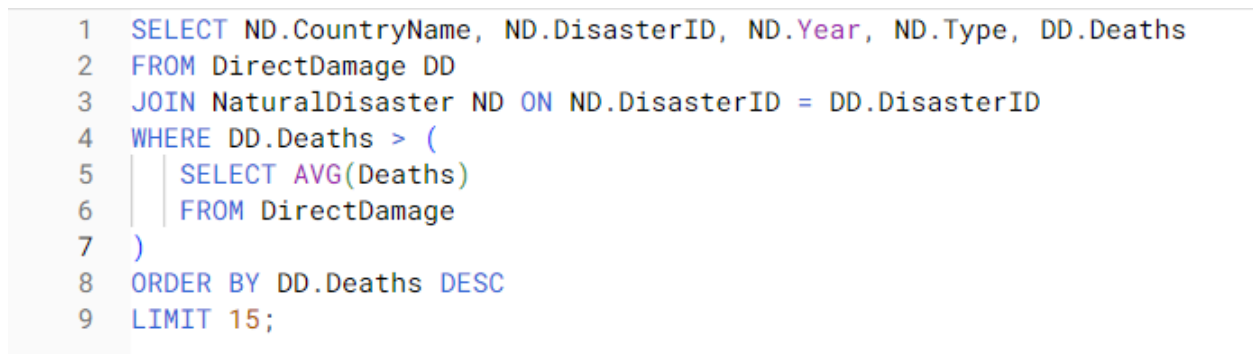
```
1 SELECT ND.Type, AVG(NEI.GDPAnnualPercentGrowth) AS Avg_GDP_Growth
2 FROM NaturalDisaster ND
3 JOIN NationalEconomicImpact NEI
4 ON ND.CountryName = NEI.CountryName AND ND.Year = NEI.Year
5 GROUP BY ND.Type;
```

Below the query editor, there is a section titled 'RESULTS'. It contains 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 interface. The query text is as follows:

```
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:

<pre>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;</pre>												
RESULTS												
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	ND		ALL					10431	100.00	Using where; ...	▼
1	SIMPLE	NEI		eq_ref	PRIMARY	PRIMARY	406	test_databoos.ND.CountryName,test_databoos.ND.Year	1	100.00		
Rows per page:									20 ▼	1 – 2 of 2	< < > >	

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.

Query 1, Index 1: **CREATE INDEX idx_nd_country_year ON NaturalDisaster(CountryName, Year);**

After Index:

<pre>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;</pre>												
RESULTS												
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	NEI		ALL	PRIMARY				4369	100.00	Using temporary	
1	SIMPLE	ND		ref	idx_nd_country_year	idx_nd_country_year	408	test_databoos.NEI.CountryName,test_databoos.NEI.Year	1	100.00		

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:

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 over 10,000 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 1, Index 2: **CREATE INDEX idx_nd_type_country ON NaturalDisaster(Type, CountryName);**

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;
```

RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		ALL	idx_nd_type_country				10431	100.00	Using where; Using temporary
1	SIMPLE	NEI		eq_ref	PRIMARY	PRIMARY	406	test_database.ND.CountryName,test_database.ND.Year	1	100.00	

Now, the NaturalDisaster table still performs a full table scan, even with the new index on (Type, CountryName). NEI continues to use its primary key with an eq_ref join.

Rationale:

A new index was created on the Type and CountryName columns in the NaturalDisaster table. This aimed to help the grouping by Type and potentially assist with the join. However, the EXPLAIN output shows that the NaturalDisaster table still performs a full table scan of the 10,431 rows, meaning that the new index was not used. The NEI table continues to be accessed using its composite primary key (CountryName, Year), retrieving one row per match. Since the join condition is not based on Type, the new index did not benefit the join or grouping in this case.

Query 1, Index 3: **CREATE INDEX** idx_nd_type_year **ON** NaturalDisaster(**Type**, **Year**);

After Index:

<pre>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;</pre>											
RESULTS ⌵											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		ALL	idx_nd_type_year				10431	100.00	Using where; Using temporary
1	SIMPLE	NEI		eq_ref	PRIMARY	PRIMARY	406	test_databoose.ND.CountryName,test_databoose.ND.Year	1	100.00	

The NaturalDisaster table is still doing a full scan, even after adding the new index on Type and Year, meaning MySQL did not consider it useful for the join or grouping.

Rationale:

A third index was created on the Type and Year columns in the NaturalDisaster table. The goal was to support both the grouping on Type and the join on Year. However, the EXPLAIN output shows that the NaturalDisaster table continued performing a full table scan, processing the 10,431 rows. The new index was not used. The NEI table still used its composite primary key (CountryName, Year) to retrieve one row per match. Since Type wasn’t part of the join condition, and Year was not needed in this grouping, the index didn’t affect the optimizer’s plan.

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:

<pre>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; 10</pre>											
RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	DD		index	idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	33.33	Using where; Using index; Using filesort
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		index		idx_dd_disasterid_deaths	10		10386	100.00	Using index

Query 2, Index 1: `CREATE INDEX idx_deaths ON DirectDamage(Deaths);`

After Index:

<pre>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; 10</pre>											
RESULTS											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	DD		range	idx_dd_disasterid_deaths,idx_deaths_only	idx_deaths_only	5		513	100.00	Using where; Backward index scan
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		index		idx_deaths_only	5		10386	100.00	Using index

Now, DirectDamage is using the new index with a range scan on Deaths, which is better. NaturalDisaster continues to join efficiently using its primary key.

Rationale:

For the second query, which gets disasters with above-average deaths, the first indexing attempt added an index on just the Deaths column. Previously, the query had to examine all rows in the DirectDamage table, filtering a third of them. After applying the new idx_deaths_only index, the EXPLAIN shows that only 513 rows were scanned using a range condition, demonstrating a large improvement. The NaturalDisaster table continued to use its primary key for joining, which remained efficient. This shows how isolating the column used in the WHERE condition (Deaths > AVG) allowed targeting of the relevant rows without scanning the full dataset.

Query 2, Index 2: `CREATE INDEX idx_deaths_disasterid ON DirectDamage(Deaths, DisasterID);`

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;

10

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	DD		range	idx_dd_disasterid_deaths,idx_deaths_disasterid	idx_deaths_disasterid	5		513	100.00	Using where...
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		index		idx_dd_disasterid_deaths	10		10386	100.00	Using index

Now, DirectDamage is using the new composite index in a range scan, filtering on Deaths while also supporting the join on DisasterID.

Rationale:

For the second query, which filters disasters with above-average deaths and joins on DisasterID, a new index was created on the Deaths and DisasterID columns. The EXPLAIN output shows that the query performs a range scan using the new index, and the subquery also benefits from index access. The NaturalDisaster table continues using its primary key to match one row per join. This composite index helps filter records by Deaths while supporting join performance, reducing rows processed in the main table to 513. The new range access path shows better efficiency over previous full scans.

Query 2, Index 3: `CREATE INDEX idx_disasterid_deaths ON DirectDamage(DisasterID, 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		index	idx_dd_disasterid_deaths,idx_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	33.33	Using where...
1	PRIMARY	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoose.DD.DisasterID	1	100.00	
2	SUBQUERY	DirectDamage		index		idx_dd_disasterid_deaths	10		10386	100.00	Using index

Now, DirectDamage is using the new composite index for both filtering and joining, but performance didn't improve.

Rationale:

For the second query, this third index was created on DirectDamage(DisasterID, Deaths) to better support both the join with NaturalDisaster and filtering on the Deaths column. After indexing, EXPLAIN showed the query using the new index across both the main and subquery. The number of rows examined in DirectDamage remained high. This means that, even if the index was used, the subquery and > comparison limits performance gains. The data structure and logic still require full access for accurate filtering.

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		index	idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	100.00	Using where; Using index; Using temporary; Using filesort
1	SIMPLE	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoosse.DD.DisasterID	1	100.00	

Query 3, 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	Extra
1	SIMPLE	DD		index	idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	100.00	Using where; Using index; Using temporary; Using filesort
1	SIMPLE	ND		eq_ref	PRIMARY	PRIMARY	4	test_databoosse.DD.DisasterID	1	100.00	

Rationale:

For the third query, which calculates the average deaths by disaster type and intensity, the first index added was on the DisasterID and Deaths columns in the DirectDamage table. That said, the EXPLAIN showed that the index was already being used without creating it—the optimizer had already chosen it as the best option. The rows remained the same, and even though the new index had efficient joins, the grouping and ordering still required temporary tables and filesort. So despite the fact that the index is relevant to the query structure, it didn't result in a better performance.

Query 3, 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	Extra
1	SIMPLE	DD		index	idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	100.00	Using where; Using index; Using temporary; Using filesort
1	SIMPLE	ND		eq_ref	PRIMARY/idx_nd_type_intensity	PRIMARY	4	test_databoosse.DD.DisasterID	1	100.00	

Now, NaturalDisaster is using the primary key, meaning the new index on (Type, Intensity) was not chosen by the optimizer, likely because the join is still by DisasterID.

Rationale:

For this query, a second index was created on the NaturalDisaster table using (Type, Intensity) to help the GROUP BY. That didn't happen— as the EXPLAIN shows, the MySQL still uses the PRIMARY key for the join condition because it happens on DisasterID. As the column isn't a part of the new index, it was not used, and the grouping remains creating a temporary table and using filesort.

Query 2, Index 3: `CREATE INDEX idx_nd_intensity_type ON NaturalDisaster(Intensity, Type);`

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	Extra
1	SIMPLE	DD		index	idx_dd_disasterid_deaths	idx_dd_disasterid_deaths	10		10386	100.00	Using where; Using index; Using temporary; Using filesort
1	SIMPLE	ND		eq_ref	PRIMARY/idx_nd_type_intensity	PRIMARY	4	test_databoosse.DD.DisasterID	1	100.00	

The query still performs a full scan on DirectDamage with the existing index. The NaturalDisaster table uses its primary key with no added benefit from the new composite index.

Rationale:

Here, a new index on the columns Intensity and Type in the NaturalDisaster table was created. This was to test if reversing the order of columns in the previous index might affect the performance of the GROUP BY clause. However, the EXPLAIN output showed that the query execution plan remained unchanged. The DirectDamage table continues to use the existing index on

DisasterID and Deaths, and the NaturalDisaster table still joins with its primary key. Again, the additional index did not improve performance in this case, not did it change how the rows were accessed.

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:

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

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		ALL					10431	11.11	Using where; Using temporary; Using filesort
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databaseo.ND.CountryName,test_databaseo.ND.Year	1	100.00	

Query 4, Index 1: CREATE INDEX idx_nd_country_only ON NaturalDisaster(CountryName);

After Index:

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

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		range	idx_nd_country_only	idx_nd_country_only	403		875	100.00	Using index condition; Using wh...
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databaseo.ND.CountryName,test_databaseo.ND.Year	1	100.00	

Now, NaturalDisasyter is using the new index with a range scan on CountryName. This drops the number of scanned rows all the way to 875, which means a significantly more efficient filtering.

Rationale:

For the fourth query, the initial EXPLAIN showed that the NaturalDisaster table performed a full scan of all 10,431 rows, even though the filter was just for ‘Indonesia’. To improve this, an index was created on the CountryName column. After adding this index, EXPLAIN shows that a range scan is being used instead, accessing only 875 rows. This represents higher efficiency, as only around 8% of the table is now scanned.

Query 4, Index 2: CREATE INDEX idx_nd_country_year ON NaturalDisaster(CountryName, Year);

After Index:

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

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; Using index; Using temporary; Using filesort
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databoose.ND.CountryName,test_databoose.ND.Year	1	100.00	

NaturalDisaster is now using the new index with a range scan, reducing scanned rows, while SEI remains having an efficient join with its primary key.

Rationale:

This is an index on the CountryName and Year columns of the NaturalDisaster table. The goal was to optimise the join and filter conditions. Pre-indexing, the query scanned all 10,431 rows in NaturalDisaster, which wasn’t good as only around 11% of them matched Indonesia. After adding the index, the EXPLAIN shows that only 875 rows were scanned, like the last index attempt, which is still a huge improvement. SEI still uses its composite primary key. Although file sorting and temporary tables are still involved, the query is more efficient now due to this index.

Query 4, Index 3: CREATE INDEX idx_nd_year_country ON NaturalDisaster(Year, CountryName);

After Index:

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

RESULTS

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	ND		index	idx_nd_year_country	idx_nd_year_country	408		10431	11.11	Using where; Using index; Using temporary; Using filesort
1	SIMPLE	SEI		eq_ref	PRIMARY	PRIMARY	406	test_databoose.ND.CountryName,test_databoose.ND.Year	1	100.00	

Now, NaturalDisaster is still using an index with a range join, but the column order has been flipped. This change does not improve performance— the rows are the same and as is the strategy.

Rationale:

For the third index in the fourth query, we tried to flip the order of the composite index column in order to match the GROUP BY and ORDER BY. We thought that MySQL might be able to traverse the index in a more efficient manner when actually grouping the data if the data is already sorted by year. According to the EXPLAIN output, this didn't change, even if the index itself was used. The optimizer had treated it similarly to the index `idx_country_year`. The temporary tables and the filesorts were still needed for grouping and ordering, meaning the performance experienced no change.