Query Optimisation Lab

Name: Divin Jacob
Student ID:

Task 1: Query Analysis

Databases Used

For this task, the queries will be run on various database sizes. The original Mondial database, the database with 10 times the entries and 20 times the entries.

Time Measurement Used

Query Pair 1: Auto Index Optimisation

SQLite is able to optimise queries using automatic indices, which can significantly improve performance by avoiding full table scans during lookups or joins. In this query pair, one query will be optimisable where SQLite can create an index, while the other will not be optimisable due to the use of transformations on indexed columns.

The queries will select cities with a population greater than 1,000,000, along with the country that the city belongs to.

Query 1A: Optimisable by using automatic index

```
SELECT c.Name AS country, ci.Name AS city
FROM Country c
JOIN City ci ON ci.Country = c.Code
WHERE ci.Population > 1000000;
```

This query would be optimisable by SQLite, as the optimiser would create an auto index on the country table when joining the city table.

Query 1B: Transformation on JOIN Column - Not Optimisable

```
SELECT c.Name AS country, ci.Name AS city
FROM Country c
JOIN City ci ON CAST(ci.Country AS TEXT) = CAST(c.Code AS TEXT)
WHERE ci.Population > 1000000;
```

This query is similar to query 1A, except that both columns in the JOIN condition are type-cast to TEXT. SQLite cannot create an auto index for this query. Indices only work on raw column values, and so this query would not be optimisable since the CAST function transforms the two columns.

Hypothesis

SQLite will optimise query 1A by using an automatic index on the `Country.Code` column for the `JOIN` condition and possibly on `City.Population` for filtering. This will result in an efficient query plan with reduced execution time.

In query 1B, because of the use of `CAST` in the `JOIN` condition, SQLite will not be able to create any indices on `Country.Code` or `City.Country`. Instead, it will perform full table scans on both tables, leading to a less efficient query plan with higher execution time.

Results and Discussion

Table 1.1: Run times on Mondial Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 1A	0.047	0.046	0.046	0.047	0.046	0.0464
Query 1B	0.056	0.053	0.055	0.054	0.054	0.0544

Table 1.2: Run times on Mondial10 Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 1A	0.067	0.067	0.065	0.066	0.065	0.0660
Query 1B	0.806	0.805	0.800	0.814	0.808	0.8066

Table 1.3: Run times on Mondial20 Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 1A	0.124	0.119	0.120	0.123	0.119	0.1210
Query 1B	3.098	3.077	3.064	3.105	3.112	3.0912

Query Pair 2: Subquery Optimisation

This query pair will investigate how SQLite optimises nested subqueries. Both queries are designed to select the names of countries alongside the average population of their cities. The goal of this analysis is to compare two different approaches to achieving the same result: one utilising a correlated subquery in the `SELECT` clause, and the other employing a subquery in the `FROM` clause

Query 2A: Optimisable by Query Flattening or Co-Routines

```
SELECT c.Name AS CountryName, s.AvgPop AS AverageCityPopulation
FROM Country c, (
    SELECT ci.Country, AVG(ci.Population) AS AvgPop
    FROM City ci
    GROUP BY ci.Country
) s
WHERE c.Code = s.Country;
```

This query consists of a subquery in the `FROM` clause, which groups the `City` table by `Country` and calculates the average city population. The result of the subquery is treated as a derived table (`s`). The outer `SELECT` statement then joins the `Country` table (`c`) with this derived table on the `Code` field of the `Country` table and the `Country` field of the derived table. The query returns the name of each country along with the corresponding average population of its cities.

Query 2B: Correlated Subquery - Not Optimisable

```
SELECT c.Name AS CountryName,
       (SELECT AVG(Population)
        FROM City
        WHERE City.Country = c.Code) AS AverageCityPopulation
FROM Country c;
```

This query uses a correlated subquery within the `SELECT` clause. For each row in the `Country` table, the subquery calculates the average population of cities in that country by referencing the `Country` table's `Code` field in the `WHERE` clause. The result is a list of countries with their respective average city populations.

Hypothesis

The SQLite optimiser should be able to optimise Query 2A into a more efficient form through query flattening or co-routine execution. The optimiser may attempt to flatten the subquery into the outer query, transforming it into a `JOIN` operation. This would eliminate the need for the subquery to be executed as a separate operation and allow for more efficient retrieval of the results. Alternatively, SQLite might choose to implement the subquery as a co-routine, which runs in parallel with the outer query, providing rows one at a time as they are needed. Using a co-routine is more memory efficient than materialising the full result of the subquery into a temporary table, and it allows the outer query to begin processing results without waiting for the entire subquery to complete.

For Query 2B, SQLite should not be able to apply the same optimisations as in Query 2A. The optimiser will likely evaluate the correlated subquery for each row of the outer query. Since this query structure does not lend itself to query flattening or co-routine execution as easily, the subquery may be evaluated repeatedly for each row in the `Country` table, which would mean that the `City` table is scanned for each country - this should lead to a higher runtime than Query 2A.

Results and Discussion

Table 2.1: Run times on Mondial Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 2A	0.048	0.046	0.046	0.046	0.047	
Query 2B	0.050	0.049	0.049	0.047	0.048	

Table 2.2: Run times on Mondial10 Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 2A	0.059	0.061	0.059	0.061	0.059	
Query 2B	5.266	5.245	5.248	5.306	5.353	

Query Pair 3: JOIN Order Optimisation

The SQLite optimiser is able to optimise queries with multiple JOINs by reordering them. This query pair will explore SQLite's join reordering optimisation and its behaviour with CROSS JOINs. SQLite's optimiser aims to reorder joins to reduce the computational cost of a query. While it can reorder INNER JOINs freely, it handles CROSS JOINs and outer joins differently, typically not allowing reordering for CROSS JOINs. The queries use different join strategies to retrieve countries (with a population greater than 10 million), with their capitals, and the continent the countries encompass (with continent area being larger than 30 million km squared).

Query 3A: JOIN - JOIN Order Optimisable

```
SELECT Country.Name AS Country, City.Name AS CapitalCity,
    City.Population AS CapitalPopulation, Continent.Name AS Continent
FROM Country
JOIN encompasses ON encompasses.Country = Country.Code
JOIN Continent ON encompasses.Continent = Continent.Name
JOIN City ON City.Name = Country.Capital AND City.country = Country.Code
WHERE Continent.Area > 30000000 AND Country.Population > 10000000;
```

This query involves (INNER) JOINs between the Country, encompasses, Continent, and City tables. The JOINs link these tables based on specific conditions, such as matching `Country.Code` with `encompasses.Country` and `City.Name` with `Country.Capital`. The `WHERE` clause filters the rows based on the country population and continent area.

Query 3B: CROSS JOIN - JOIN Order Not Optimisable

```
SELECT Country.Name AS Country, City.Name AS CapitalCity,
    City.Population AS CapitalPopulation, Continent.Name AS Continent
FROM City
CROSS JOIN encompasses
CROSS JOIN Continent
CROSS JOIN Country
WHERE Continent.Area > 30000000 AND Country.Population > 10000000
    AND encompasses.Country = Country.Code AND encompasses.Continent = Continent.Name
    AND City.Name = Country.Capital AND City.country = Country.Code;
```

In this query, CROSS JOINs are used between the City, encompasses, Continent, and Country tables. A CROSS JOIN produces a Cartesian product of all tables involved, meaning that every combination of rows from the tables is generated. The filtering conditions in the `WHERE` clause are applied, to ensure that valid rows are returned.

Hypothesis

SQLite should optimise the query 3A by selecting an efficient join order through join reordering. Since (INNER) JOINs are commutative, the optimiser can reorder the joins in the `FROM` clause to find the most efficient execution order. The query planner evaluates the cost of each potential join order, considering factors such as the table size, available indexes, and filtering conditions like `Country.Population` and `Continent.Area`. For example, the `Country.Population` filter might reduce the number of rows in the Country table early on, which could help minimise the data involved in subsequent joins.

On the other hand, Query 3B uses CROSS JOINs, which do not allow join reordering. This means that the tables must be processed in the order they appear in the FROM clause, and the optimiser would not be able to adjust the nesting order to minimise work. This approach should lead to larger intermediate results and require more processing time, resulting in a less efficient query plan compared to Query 3A.

Results and Discussion

Table 3.1: Run times on Mondial Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 3A	0.079	0.080	0.076	0.084	0.079	
Query 3B	0.089	0.092	0.086	0.087	0.087	

Table 3.2: Run times on Mondial10 Database

Query	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run (5)	Average (s)
Query 3A	1.180	1.162	1.195	1.178	1.184	
Query 3B	7.543	7.161	7.088	7.102	7.263	