An Investigation into Query Optimisation of Semantically Equivalent Queries in SQLite3

Jonathon Dilworth
School of Computer Science
University of Manchester, United Kingdom
Email: jonathon.dilworth@postgrad.ac.uk

I. Experimental Results: Task 1

Each pair of semantically equivalent queries is listed below under its distinct section. Each section begins with a brief description of the query pair and an explanation of the expected optimisation strategies to be employed by the optimiser. Then, a set of hypotheses is stated, and experimental results are recorded. The experimental results are documented within tables and visualised through bar charts and line plots. Furthermore, their respective query execution plans are examined. Finally, each section contains an analysis and an evaluation of the observed results.

Query Pair One: Average Mountain Elevation Height

The first query pair under examination focuses on the potential effects that a subquery found within the FROM clause may have on the optimisation procedure and selected execution plan. The query pair is stated below.

QP1.1

```
SELECT AVG(Elevation)
FROM
(
    SELECT Elevation FROM Mountain
);
```

QP1.2

```
SELECT AVG(Elevation)
FROM
(
    SELECT NAME, Elevation
    FROM Mountain
    INTERSECT
    SELECT Name, Elevation
    FROM Mountain
    UNION ALL
    SELECT Name, Elevation
    FROM Mountain
);
```

Explanation of the Optimisation Strategies and Effects on Evaluation

QP1.1

Whilst QP1.1 is a reasonably rudimentary example, it is expected to demonstrate the optimiser's capacity to perform subquery flattening [1]. This strategy should be employed to eliminate the need to create a transient table [1] to evaluate the outer query. The transformation associated with this optimisation strategy aims to reduce the form of the original query to something much simpler; in this instance, the subquery can be eliminated:

```
SELECT AVG(Elevation) FROM Mountain;
```

This optimisation (or rewrite) will result in a query execution plan (QEP) consisting of a single table scan rather than creating a transient table and two table scans. Theoretically, this should minimise the actual cost2 involved in executing this query by minimising the anticipated cost3. Thus reducing computational and physical overhead and placing less strain on the query evaluator.

QP1.2

In the case of QP1.2, the subquery is substantially more complex and takes the form of a compound SELECT [2] utilising operations that must satisfy both set and multiset semantics. The presence of INTERSECT is the critical factor in constraining the optimiser such that it cannot reduce this query to a simpler form through query flattening [1], as is the case in QP1.1.

Since the optimiser cannot effectively flatten this query, one may assume that the QEP will likely involve an intermediate result set. Whilst this is not necessarily the case in this instance [1], it would be reasonable to assume (and was assumed under the decision to investigate this query pair). Hence, a brief discussion of the supposed implications is deemed noteworthy. As such, it may be asserted that the cardinality of the transient table will be double the value it ought to be due to the existence of duplicates (since INTERSECT takes precedence over UNION ALL). Thus, the execution time (in terms of real-time and user-plus-system time) is almost guaranteed to be longer. The set operations necessitate the creation of temporary B-TREEs, and the assumed transient table must be held in memory. These operations should increase computational overhead regarding memory management and I/O.

Furthermore, as the dataset grows, the Mountain table's size increases proportionally to this growth, potentially leveraging additional supervisor calls on cache misses, compounding the required system time. In addition, each SELECT clause will involve an additional table scan. The effect of these suboptimal evaluation strategies will ultimately lead to increased computational complexity in both time and space.

Whilst the optimiser has a third option (which is preferable), this was only realised after running the queries. It would be disingenuous (and constitute academic malpractice) to change the original hypotheses after observing both the selected query execution plan and the experimental results. Thus, a discussion of the actual QEP (and the use of co-routines) will be found within the analysis.

Finally, it is important to note that semantic equivalence holds under all conditions for this query pair. To clarify, the existence of duplicates produced in the manner described by QP1.2 does not affect the result returned through AVG. Doubling the cardinality of any assumed transient table in the proposed manner also doubles the total elevation.

$$\frac{\sum_{i=1}^{|R|} e_i}{|R|} \equiv \frac{\sum_{i=1}^{2|R|} e_i}{2|R|}$$

Hypothesises

1) QP1.1 will outperform QP1.2.
2) The optimiser will produce a QEP involving a single table scan for
3) The QEP for QP1.2 will produce a transient table for subquery evaluation.
4) The QEP for QP1.2 will involve using temporary B-Trees to resolve set operations.
5) The QEP for QP1.2 will include multiple table scans, one for each SELECT statement.
6) QP1.2 will result in a greater system time due to increased I/O operations and utilisation of temporary B-Trees.

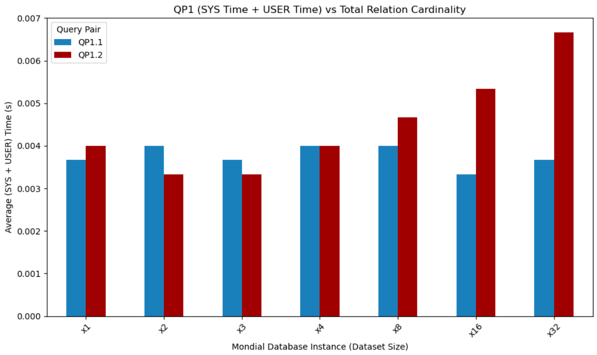
Results

Firstly, the query execution plans for each query within the pair are provided. Shortly after, the results from QP1.1 and QP1.2 are presented in two tables and visualised using two plots. The tables document the precise times recorded for each query during the experiment. These times are presented in terms of real-time and 'effective CPU time', calculated by combining the user and system time. Secondly, the plots reflect the values within the tables and allow an intuitive interpretation of the result set.

A seriously comprehensive record of experimental context, results, analysis and evaluation can be found at https://docs.google.com/spreadsheets/d/1sa_EJLCuVCU9AJDF89QFWLuktLDrhPWd9ZDx8zutFg0/edit?usp=sharing, reflected in Appendix A, B and C.

QP1.1 Query Execution Plan

```
QUERY PLAN
`--SCAN Mountain
```



QP1.2 Query Execution Plan

```
QUERY PLAN
|--CO-ROUTINE (subquery-3)
|   `--COMPOUND QUERY
|       |--LEFT-MOST SUBQUERY
|       |   `--SCAN Mountain
|       |--INTERSECT USING TEMP B-TREE
|       |   `--SCAN Mountain
|       `--UNION ALL
|           `--SCAN Mountain
`--SCAN (subquery-3)
```

Table I
AVERAGE (USER + SYSTEM) TIME MEASURES (SECONDS)

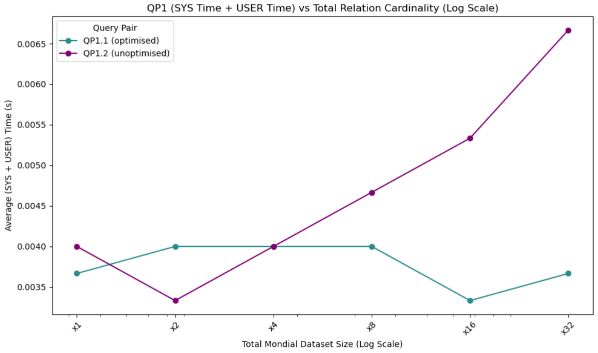
DB Instance	QP1.1	QP1.2	Delta (%)
mondial1.db	0.003667	0.004	9.08
mondial2.db	0.004	0.003333	-16.68
mondial3.db	0.003667	0.003333	-9.11
mondial4.db	0.004	0.004	0.0
mondial8.db	0.004	0.004667	16.68
mondial16.db	0.003333	0.005334	60.04
mondial32.db	0.003667	0.006667	81.81

Table II
AVERAGE REAL TIME MEASURES (SECONDS)

DB Instance	QP1.1	QP1.2	Delta
mondial1.db	0.005	0.005333	6.66%
mondial2.db	0.005	0.005	0.0%
mondial3.db	0.005333	0.005333	0.0%
mondial4.db	0.005	0.005	0.0%
mondial8.db	0.005	0.006	20.0%
mondial16.db	0.004667	0.007	49.99%
mondial32.db	0.005333	0.008	50.01%

Analysis and Evaluation

While most of the hypotheses stated were proven correct, the most significant ones do not appear to hold under all conditions, thus disproving them. QP1.1 does not outperform QP1.2 in all cases for this experiment, which is surprising. However, this can only be stated for relatively small relation



cardinalities and when performance is measured strictly in terms of user and system time. For instance, the 'effective CPU execution time' for QP1.2 on mondial2.db took 3.33ms as compared to QP1.1, having taken 4ms, demonstrating an 18.28% difference in absolute value and a 20.12% increase in performance (or relative delta) [3] [4]. Moreover, as the dataset size (and consequently, the size of the Mountain table) grows exponentially, the optimiser struggles to mitigate the performance limitations of computing large set operations within the given subquery. Under these conditions, H4, H5 and H6 are most apparent. Still, the unexpected use of co-routines is doing an admirable job of maintaining a respectable degree of evaluator efficiency.

The most significant shortcoming in the initial formulation of this query pair was an oversight regarding the optimiser's utilisation of co-routines during query execution planning. Having poured over the SQLite documentation, it is challenging to justify why the results suggest that utilising co-routines that involve three table scans may be as efficient or more efficient than a single table scan for small datasets. Initial intuition would suggest that the results could be inaccurate. However, some evidence suggests that co-routines (albeit under evidently different conditions) can perform highly efficiently on small datasets [5]. In addition, the notion of utilising co-routines to allow for non-blocking, incremental data processing within the same thread may hint at the usefulness of this mechanism for optimisation. Regardless, this area requires further in-depth review of the literature surrounding database systems and CPU architecture. Given what has been read thus far, different resources discuss co-routines in varying contexts, and the information available on the official SQLite website does not necessarily corroborate these alternative sources of information [6] [5].

Finally, a series of additional observations can be made through comprehensive experimental analysis. For instance, the absolute difference in 'effective CPU execution time' across the entire query pair for all database instances is measured as 0.861 ms, representing a 22.96% increase in average performance (see Appendix A, B and C).

Query Pair Two

The second pair of semantically equivalent queries aims to measure SQLite's optimiser performance and the evaluator's execution time by focusing on the use of DISTINCT (and somewhat unnecessarily, an ORDER BY clause, as the two queries are semantically equivalent under multiset semantics regardless of order [7]) versus a complex JOIN that employs GROUP BY.

QP2.1

```
SELECT DISTINCT Name, Country
FROM City
ORDER BY Name;
```

QP2.2

```
SELECT City.Name, City.Country
FROM City AS c1
JOIN City ON c1.Name = City.Name
WHERE c1.Country = City.Country
GROUP BY City.Name, City.Country;
```

Explanation of the Optimisation Strategies and Effects on Evaluation

QP2.1

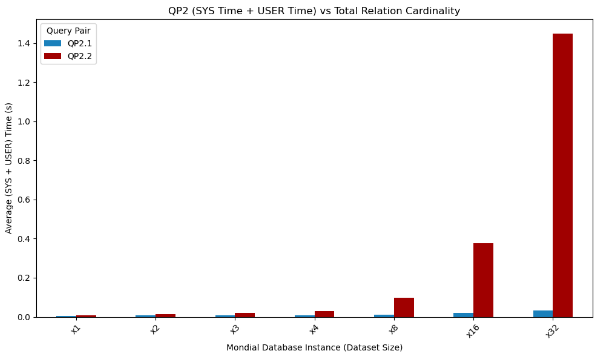
Again, QP2.1 is a fairly simple example (by design) that aims to use temporary B-trees for optimal performance, removing duplicates and dictating the final result set order (which happens to be implicit in QP2.2, though QP2.2 always results in an equivalent ordering to QP2.1). The query execution plan should result in a single table scan but will also include the creation of temporary B-trees for the abovementioned reasons.

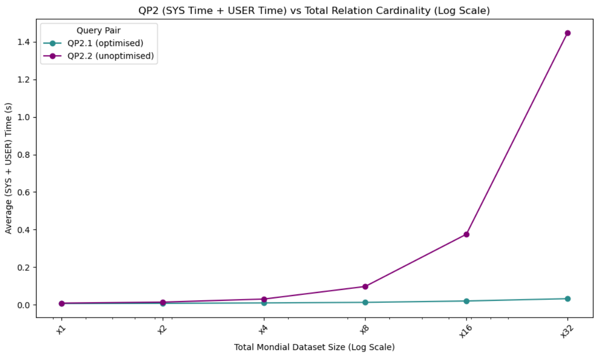
QP2.2

There is a lot more to QP2.2; whilst it shares some similarities in sub-optimal strategies that are likely to be adopted with the previous query pair (such as a sizeable, but arguably unnecessary, intermediate result size), it also introduces novel mechanisms that still need to be discussed within this report. Specifically, it should automatically create an index due to multiple SELECT attributes across an implicit INNER JOIN. Taken together with aggregation as a means of eliminating duplicates through GROUP BY, it is anticipated that the resulting query execution plan will be distinctly different and require significantly more I/O (due to the complexities of processing an aggregate over the JOIN). Thus, QP2.1's performance should be vastly superior to QP2.2's.

Hypothesises

1) QP2.1 will maintain a fairly constant performance with minimal
2) QP2.2 will significantly underperform on larger datasets.
3) QP2.2 will exhibit exponential growth in all performance metrics as the dataset increases in size.





Results

QP2.1 Query Execution Plan

```
QUERY PLAN
|--SCAN City
|--USE TEMP B-TREE FOR DISTINCT
`--USE TEMP B-TREE FOR ORDER BY
```

QP2.2 Query Execution Plan

```
QUERY PLAN
|--SCAN c1
|--SEARCH City USING AUTOMATIC COVERING
|   INDEX (Name=? AND Country=?)
`--USE TEMP B-TREE FOR GROUP BY
```

Table III
AVERAGE (USER + SYSTEM) TIME MEASURES (SECONDS)

DB Instance	QP1.1	QP1.2	Delta
mondial1.db	0.006	0.008	33.33%
mondial2.db	0.007	0.013333	90.47%
mondial3.db	0.008	0.021	162.5%
mondial4.db	0.009	0.029667	229.63%
mondial8.db	0.012	0.096667	705.56%
mondial16.db	0.019334	0.375	1839.59%
mondial32.db	0.031667	1.449333	4476.79%

Table IV
AVERAGE REAL TIME MEASURES (SECONDS)

DB Instance	QP1.1	QP1.2	Delta
mondial1.db	0.007667	0.009333	21.73%
mondial2.db	0.008333	0.015333	84.0%
mondial3.db	0.009	0.023333	159.26%
mondial4.db	0.01	0.031	210.0%
mondial8.db	0.014	0.1	614.29%
mondial16.db	0.020667	0.417667	1920.94%
mondial32.db	0.033667	1.462	4242.53%

Analysis and Evaluation

Given that all hypothesises were proven to be correct and that the query execution plans seemed to match what was expected, it can be confidently stated that the introduction of a JOIN (albeit a somewhat redundant operation from the programmer's perspective in this instance) and associated aggregate function GROUP_BY causes the optimiser a great deal of difficulty when serving an efficient QEP to the evaluator.

The execution time demonstrates linear growth in QP2.1 and exponential growth in all performance metrics concerning QP2.2. This growth rate is exemplified through the experimental result data showing QP2.1 taking 6ms through 31.67ms (user plus sys time) to finish execution on instances of mondial1.db through mondial32.db, versus QP2.2, having taken 8ms through 1449.33ms for the same database instances. This huge difference in performance represents an average absolute difference of 314.9ms measured across all seven database instances, resulting in an average relative delta of 2209.91%, which is significant (see Appendix A, B and C).

Query Pair Three

The third and final query pair contains syntactically equivalent queries to be run under different conditions. In the first instance, they will be run under an index, and in the second, they will run without an index.

QP3 Create Index

```
CREATE INDEX IF NOT EXISTS idx_n_c_p
ON City (Name, Country, Province);
```

QP3 Drop Index

```
DROP INDEX IF EXISTS idx_n_c_p;
```

QP3.1 & QP3.2

```
SELECT Name, Province
FROM City
WHERE Province = 'Nordrhein-Westfalen';
```

Explanation of the Optimisation Strategies and Effects on Evaluation

Given the nature of the query requiring both Name and Province, QP3.1 should be able to effectively search the index, thereby avoiding a full table scan and resulting in a significant performance gain. In contrast, QP3.2 will be forced to perform a full table scan to evaluate the expression within the WHERE clause, resulting in poorer performance, especially as the dataset size grows.

Hypothesises

1) Since QP3.1 operates with the specified index, it should consistently outperform QP3.2, operating without the aforementioned index.
2) The relative delta in performance (execution time) should drastically increase as the dataset grows.
3) QP3.1 will utilise a search of the index, whereas QP3.2 will not be capable of doing so; this will be reflected in the query execution plans.

Results

QP3.1 Query Execution Plan

```
QUERY PLAN
`--SCAN City USING
   COVERING INDEX idx_n_c_p
```

QP3.2 Query Execution Plan

```
QUERY PLAN
`--SCAN City
```

Table V
AVERAGE (USER + SYSTEM) TIME MEASURES (SECONDS)

DB Instance	QP1.1	QP1.2	Delta
mondial1.db	0.002	0.002	0.0%
mondial2.db	0.002333	0.002333	0.0%
mondial3.db	0.002	0.002333	16.65%
mondial4.db	0.002	0.002	0.0%
mondial8.db	0.003333	0.003333	0.0%
mondial16.db	0.005	0.004333	-13.34%
mondial32.db	0.006	0.007333	22.22%

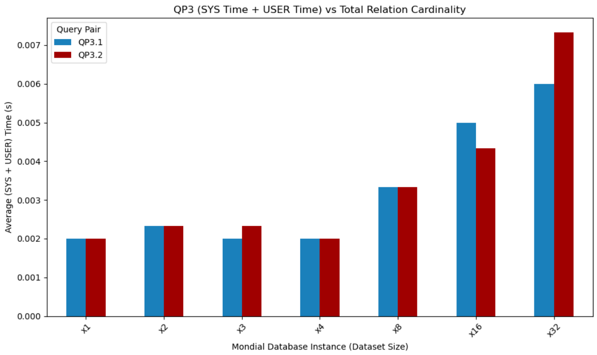
Table VI
AVERAGE REAL TIME MEASURES (SECONDS)

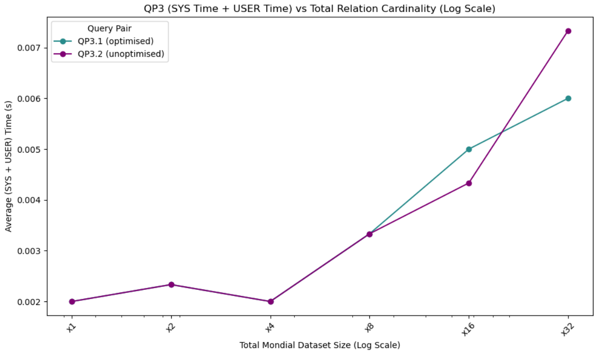
DB Instance	QP1.1	QP1.2	Delta
mondial1.db	0.003	0.003333	11.1%
mondial2.db	0.004333	0.006	38.47%
mondial3.db	0.003667	0.004333	18.16%
mondial4.db	0.004	0.005	25.0%
mondial8.db	0.004667	0.005	7.14%
mondial16.db	0.006	0.005667	-5.55%
mondial32.db	0.008	0.009	12.5%

Analysis and Evaluation

The results invalidate all of the hypotheses except H3, which is, again, surprising. They indicate that the index is ineffective when operating over the various instances of Mondial, as there is negligible variance in the recorded execution times. When the real execution time is considered, there appears to be a performance gain for the query running under an index initially. However, the performance gain is negligible as the dataset increases in cardinality. Thus, any initial gain could be explained as background processes introducing variance in real-time measures.

For instance, when examining the real-time performance of mondial16,db QP3.2 outperforms QP3.1 by 0.33ms, representing a 5.88% increase in the relative delta. Given the experimental results, it is difficult to make any strong claims whatsoever.





II. EXECUTION ENVIRONMENT: TASK 2

Hardware

1) Physical Machine: Apple MacBook Pro, 2023.
2) Processor: Apple M2 Max.
3) Unified Memory: 64GB.

Software

1) Operating System: macOS Sonoma 14.61.
2) SQLite Version: v3.43.2.

Database Instances: State & Size

There are no explicit keys of any kind defined within any DB instance on production from dbfactory.sh:

1) No Indexes
2) No Primary Keys
3) No Foreign Keys

4) No Composite Keys

There are minimal constraints, however the following are to be found within the schema, and as such, are present:

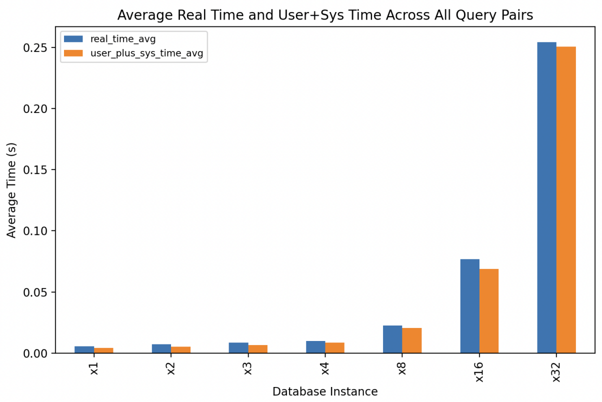
1) Country.Name IS NOT NULL
2) encompasses.Country IS NOT NULL
3) encompasses.Couninent IS NOT NULL
4) isMember.Type DEFAULT ’member’
5) Organization.Name IS NOT NULL
6) Province.Name IS NOT NULL
7) Province.Country IS NOT NULL

Table VII
DATABASE INSTANCE INFORMATION

Database Instance	Relation Scaling	$\sum$ Relation Cardinality
mondial1.db	x1	42,323
mondial2.db	x2	84,646
mondial3.db	x3	126,969
mondial4.db	x4	169,292
mondial8.db	x8	338,584
mondial16.db	x16	677,168
mondial32.db	x32	1,354,336

Project File Structure

```
.
|-- data
| |-- databases
| | |-- mondial1.db
| | |-- ...
| | |-- mondial32.db
| |-- sql_scripts
|   |-- MondialCreateNoKeys.sql
|   |-- ...
|   |-- MondialCreate32Times.sql
|   |-- mondial-create-tables-no-keys.sql
|   |-- mondial-populate.sql
|-- results
| |-- csv
| | |-- results.csv
| | |-- indexed.csv
| | |-- unindexed-timings.csv
| |-- logs
| |   |-- QP_N_DB_X_*.log
| |-- stdout
| |   |-- QP_N_DB_X_*.out
| |-- visualisations
| | |-- real_time
| | | |-- *.png
| | |-- user_plus_sys_time
| |   |-- *.png
| |-- write_up
|     |-- final_report
|     | |-- report.pdf
|     | |-- *.tex
|     |-- tables
|     | |-- latex_tables
|     | | |-- QPN_*.tex
|     | |-- csv
|     |   |-- QPN_*.csv
|     |-- spreadsheets-for-analysis
|         |-- FULL-DATASET-EXCEL.xlsx
|         |-- FULL-DATASET.ods
|         |-- FULL-DATASET-WITH-HEADERS.csv
|         |-- backup.ods
|-- scripts
| |-- dbfactory.sh
| |-- load_and_run_sqlite_script.sh
| |-- mondial_relation_sizes.py
| |-- produce_visualisations.py
|-- src
    |-- queries
        |-- qp1_optimised.sql
        |-- ...
        |-- qp3_unoptimised.sql
```



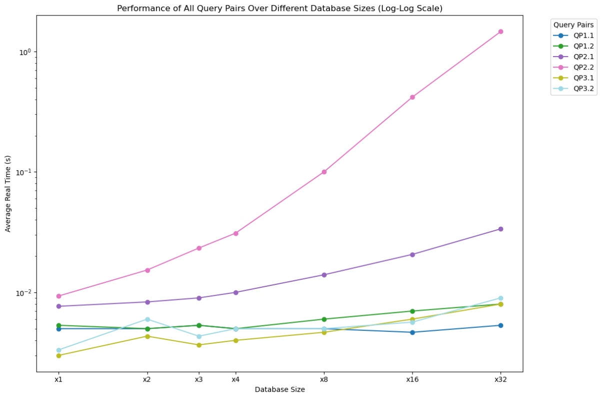
III. DISCUSSION: TASK 2

Data Visualisation: Real-Time, User-Time & Sys-Time

Presenting all of the data contained within this report in a single plot proved somewhat challenging; as such, two plots have been produced, the first attempts to better understand the relationship between real-time and (sys+user) time, what has been referred to within this report as ’effective CPU execution time’. Whilst the two metrics are correlated within the context of this report (given the associated plot within this section), the physical hardware on which the experiments are run may yield vastly different results.

Further Reading: Co-routines

A co-routine may be called only when necessary, so it can be considered an optimisation in many instances. For example, when the outer query provides a LIMIT BY clause, the rewriter — or query optimiser — may be able to apply that clause to the inner query conditionally. Performing such a rewrite would



be advantageous as it would relieve the evaluator from having to compute the entire sub-query before evaluating the outer query, essentially ’pruning’ the execution process. This ability to prune, combined with minimising latency associated with reading from disk, may further hint at how QP1.2 performed so well. However, the documentation available through SQLite’s website seems to suggest that the caller and callee are co-dependant, which was initially taken to mean that they are synchronous and blocking operations, though, having done some additional reading, it is difficult to know for sure (at present, given current time constraints) how exactly these mechanisms are implemented and how the use of them varies across hardware platforms. I will continue to research this area of database systems, but it is likely beyond the scope of this assignment for the time being.

References

[1] “The SQLite Query Optimizer Overview.” [Online]. Available: https://www.sqlite.org/optoverview.html

[2] “SELECT.” [Online]. Available: https://www.sqlite.org/lang_select.html

[3] “Relative change,” Nov. 2024, page Version ID: 1259165010. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Relative_change&oldid=1259165010

[4] C. LLC, “Percentage Difference Calculator.” [Online]. Available: https://www.calculatorsoup.com/calculators/algebra/percent-difference-calculator.php

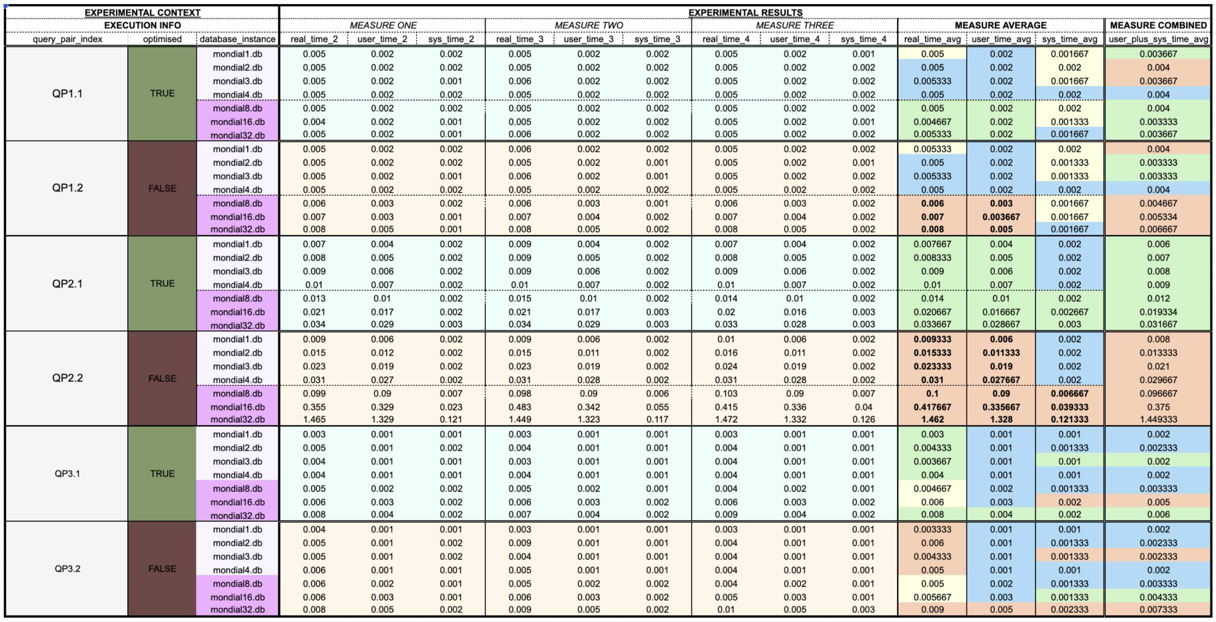
[5] C. Jonathan, U. F. Minhas, J. Hunter, J. Levandoski, and G. Nishanov, “Exploiting coroutines to attack the "killer nanoseconds",” *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1702–1714, Jul. 2018. [Online]. Available: https://dl.acm.org/doi/10.14778/3236187.3236216

[6] “terminology - Difference between subroutine , co-routine , function and thread? - Stack Overflow.” [Online]. Available: https://stackoverflow.com/questions/24780935/difference-between-subroutine-co-routine-function-and-thread

[7] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, “Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries,” *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1482–1495, Jul. 2018. [Online]. Available: https://dl.acm.org/doi/10.14778/3236187.3236200

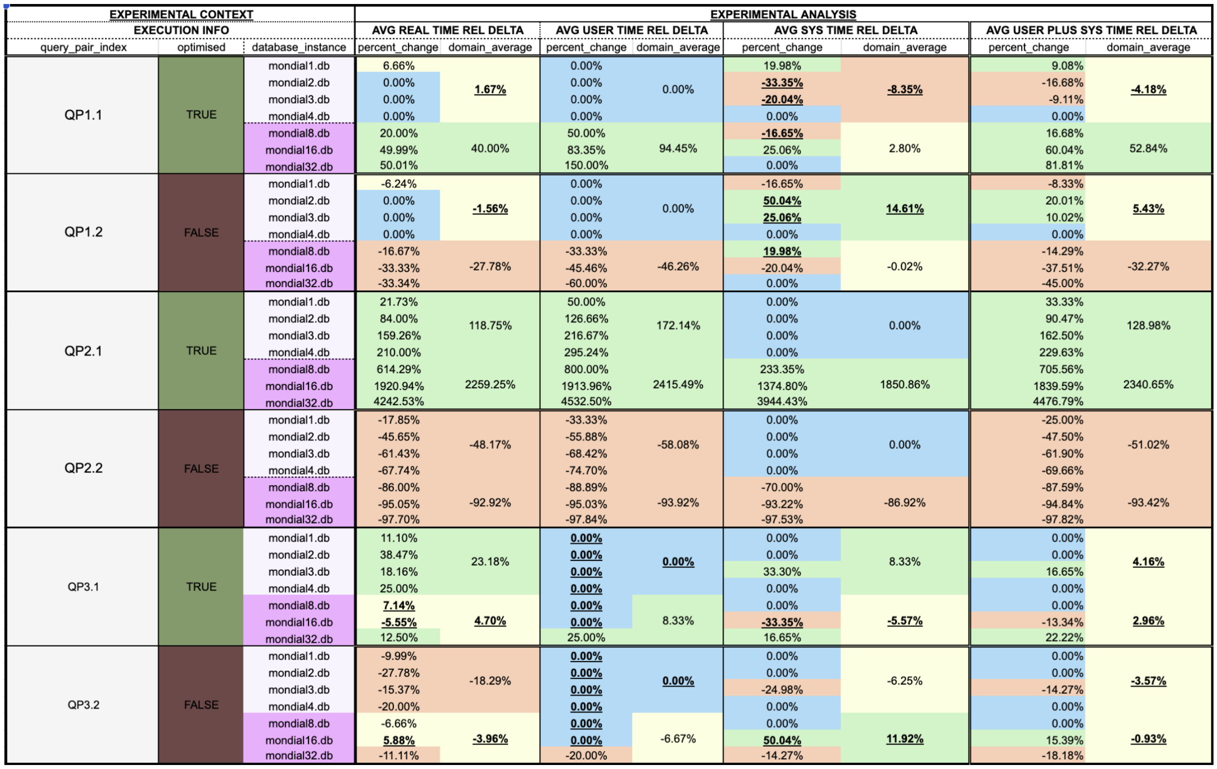
Appendix

Appendix A



EXPERIMENTAL CONTEXT			EXPERIMENTAL RESULTS												
EXECUTION INFO			*MEASURE ONE*			*MEASURE TWO*			*MEASURE THREE*			MEASURE AVERAGE			MEASURE COMBINED
query_pair_index	optimised	database_instance	real_time_2	user_time_2	sys_time_2	real_time_3	user_time_3	sys_time_3	real_time_4	user_time_4	sys_time_4	real_time_avg	user_time_avg	sys_time_avg	user_plus_sys_time_avg
QP1.1	TRUE	mondial1.db	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.001	0.005	0.002	0.001667	0.003667
		mondial2.db	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.004
		mondial3.db	0.005	0.002	0.001	0.006	0.002	0.002	0.005	0.002	0.002	0.005333	0.002	0.001667	0.003667
		mondial4.db	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.004
		mondial8.db	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.004
		mondial16.db	0.004	0.002	0.001	0.005	0.002	0.002	0.005	0.002	0.001	0.004667	0.002	0.001333	0.003333
		mondial32.db	0.005	0.002	0.001	0.006	0.002	0.002	0.005	0.002	0.002	0.005333	0.002	0.001667	0.003667
QP1.2	FALSE	mondial1.db	0.005	0.002	0.002	0.006	0.002	0.002	0.005	0.002	0.002	0.005333	0.002	0.002	0.004
		mondial2.db	0.005	0.002	0.002	0.005	0.002	0.001	0.005	0.002	0.001	0.005	0.002	0.001333	0.003333
		mondial3.db	0.005	0.002	0.001	0.006	0.002	0.001	0.005	0.002	0.002	0.005333	0.002	0.001333	0.003333
		mondial4.db	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.005	0.002	0.002	0.004
		mondial8.db	0.006	0.003	0.002	0.006	0.003	0.001	0.006	0.003	0.002	**0.006**	**0.003**	0.001667	0.004667
		mondial16.db	0.007	0.003	0.001	0.007	0.004	0.002	0.007	0.004	0.002	**0.007**	**0.003667**	0.001667	0.005334
		mondial32.db	0.008	0.005	0.001	0.008	0.005	0.002	0.008	0.005	0.002	**0.008**	**0.005**	0.001667	0.006667
QP2.1	TRUE	mondial1.db	0.007	0.004	0.002	0.009	0.004	0.002	0.007	0.004	0.002	0.007667	0.004	0.002	0.006
		mondial2.db	0.008	0.005	0.002	0.009	0.005	0.002	0.008	0.005	0.002	0.008333	0.005	0.002	0.007
		mondial3.db	0.009	0.006	0.002	0.009	0.006	0.002	0.009	0.006	0.002	0.009	0.006	0.002	0.008
		mondial4.db	0.01	0.007	0.002	0.01	0.007	0.002	0.01	0.007	0.002	0.01	0.007	0.002	0.009
		mondial8.db	0.013	0.01	0.002	0.015	0.01	0.002	0.014	0.01	0.002	0.014	0.01	0.002	0.012
		mondial16.db	0.021	0.017	0.002	0.021	0.017	0.003	0.02	0.016	0.003	0.020667	0.016667	0.002667	0.019334
		mondial32.db	0.034	0.029	0.003	0.034	0.029	0.003	0.033	0.028	0.003	0.033667	0.028667	0.003	0.031667
QP2.2	FALSE	mondial1.db	0.009	0.006	0.002	0.009	0.006	0.002	0.01	0.006	0.002	**0.009333**	**0.006**	0.002	0.008
		mondial2.db	0.015	0.012	0.002	0.015	0.011	0.002	0.016	0.011	0.002	**0.015333**	**0.011333**	0.002	0.013333
		mondial3.db	0.023	0.019	0.002	0.023	0.019	0.002	0.024	0.019	0.002	**0.023333**	**0.019**	0.002	0.021
		mondial4.db	0.031	0.027	0.002	0.031	0.028	0.002	0.031	0.028	0.002	**0.031**	**0.027667**	0.002	0.029667
		mondial8.db	0.099	0.09	0.007	0.098	0.09	0.006	0.103	0.09	0.007	**0.1**	**0.09**	**0.006667**	0.096667
		mondial16.db	0.355	0.329	0.023	0.483	0.342	0.055	0.415	0.336	0.04	**0.417667**	**0.335667**	**0.039333**	0.375
		mondial32.db	1.465	1.329	0.121	1.449	1.323	0.117	1.472	1.332	0.126	**1.462**	**1.328**	**0.121333**	1.449333
QP3.1	TRUE	mondial1.db	0.003	0.001	0.001	0.003	0.001	0.001	0.003	0.001	0.001	0.003	0.001	0.001	0.002
		mondial2.db	0.005	0.001	0.002	0.004	0.001	0.001	0.004	0.001	0.001	0.004333	0.001	0.001333	0.002333
		mondial3.db	0.004	0.001	0.001	0.003	0.001	0.001	0.004	0.001	0.001	0.003667	0.001	0.001	0.002
		mondial4.db	0.004	0.001	0.001	0.004	0.001	0.001	0.004	0.001	0.001	0.004	0.001	0.001	0.002
		mondial8.db	0.005	0.002	0.002	0.005	0.002	0.001	0.004	0.002	0.001	0.004667	0.002	0.001333	0.003333
		mondial16.db	0.006	0.003	0.002	0.006	0.003	0.002	0.006	0.003	0.002	0.006	0.003	0.002	0.005
		mondial32.db	0.008	0.004	0.002	0.007	0.004	0.002	0.009	0.004	0.002	0.008	0.004	0.002	0.006
QP3.2	FALSE	mondial1.db	0.004	0.001	0.001	0.003	0.001	0.001	0.003	0.001	0.001	0.003333	0.001	0.001	0.002
		mondial2.db	0.005	0.001	0.002	0.009	0.001	0.001	0.004	0.001	0.001	0.006	0.001	0.001333	0.002333
		mondial3.db	0.005	0.001	0.002	0.004	0.001	0.001	0.004	0.001	0.001	0.004333	0.001	0.001333	0.002333
		mondial4.db	0.006	0.001	0.001	0.005	0.001	0.001	0.004	0.001	0.001	0.005	0.001	0.001	0.002
		mondial8.db	0.006	0.002	0.001	0.005	0.002	0.002	0.004	0.002	0.001	0.005	0.002	0.001333	0.003333
		mondial16.db	0.006	0.003	0.001	0.006	0.003	0.002	0.005	0.003	0.001	0.005667	0.003	0.001333	0.004333
		mondial32.db	0.008	0.005	0.002	0.009	0.005	0.002	0.01	0.005	0.003	0.009	0.005	0.002333	0.007333

Appendix B



EXPERIMENTAL CONTEXT			EXPERIMENTAL ANALYSIS							
EXECUTION INFO			AVG REAL TIME REL DELTA		AVG USER TIME REL DELTA		AVG SYS TIME REL DELTA		AVG USER PLUS SYS TIME REL DELTA	
query_pair_index	optimised	database_instance	percent_change	domain_average	percent_change	domain_average	percent_change	domain_average	percent_change	domain_average
QP1.1	TRUE	mondial1.db	6.66%	**1.67%**	0.00%	0.00%	19.98%	**-8.35%**	9.08%	**-4.18%**
		mondial2.db	0.00%		0.00%		**-33.35%**		-16.68%	
		mondial3.db	0.00%		0.00%		**-20.04%**		-9.11%	
		mondial4.db	0.00%		0.00%		0.00%		0.00%	
		mondial8.db	20.00%	40.00%	50.00%	94.45%	**-16.65%**	2.80%	16.68%	52.84%
		mondial16.db	49.99%		83.35%		25.06%		60.04%	
		mondial32.db	50.01%		150.00%		0.00%		81.81%	
QP1.2	FALSE	mondial1.db	-6.24%	**-1.56%**	0.00%	0.00%	-16.65%	**14.61%**	-8.33%	**5.43%**
		mondial2.db	0.00%		0.00%		**50.04%**		20.01%	
		mondial3.db	0.00%		0.00%		**25.06%**		10.02%	
		mondial4.db	0.00%		0.00%		0.00%		0.00%	
		mondial8.db	-16.67%	-27.78%	-33.33%	-46.26%	**19.98%**	-0.02%	-14.29%	-32.27%
		mondial16.db	-33.33%		-45.46%		-20.04%		-37.51%	
		mondial32.db	-33.34%		-60.00%		0.00%		-45.00%	
QP2.1	TRUE	mondial1.db	21.73%	118.75%	50.00%	172.14%	0.00%	0.00%	33.33%	128.98%
		mondial2.db	84.00%		126.66%		0.00%		90.47%	
		mondial3.db	159.26%		216.67%		0.00%		162.50%	
		mondial4.db	210.00%		295.24%		0.00%		229.63%	
		mondial8.db	614.29%	2259.25%	800.00%	2415.49%	233.35%	1850.86%	705.56%	2340.65%
		mondial16.db	1920.94%		1913.96%		1374.80%		1839.59%	
		mondial32.db	4242.53%		4532.50%		3944.43%		4476.79%	
QP2.2	FALSE	mondial1.db	-17.85%	-48.17%	-33.33%	-58.08%	0.00%	0.00%	-25.00%	-51.02%
		mondial2.db	-45.65%		-55.88%		0.00%		-47.50%	
		mondial3.db	-61.43%		-68.42%		0.00%		-61.90%	
		mondial4.db	-67.74%		-74.70%		0.00%		-69.66%	
		mondial8.db	-86.00%	-92.92%	-88.89%	-93.92%	-70.00%	-86.92%	-87.59%	-93.42%
		mondial16.db	-95.05%		-95.03%		-93.22%		-94.84%	
		mondial32.db	-97.70%		-97.84%		-97.53%		-97.82%	
QP3.1	TRUE	mondial1.db	11.10%	23.18%	**0.00%**	**0.00%**	0.00%	8.33%	0.00%	**4.16%**
		mondial2.db	38.47%		**0.00%**		0.00%		0.00%	
		mondial3.db	18.16%		**0.00%**		33.30%		16.65%	
		mondial4.db	25.00%		**0.00%**		0.00%		0.00%	
		mondial8.db	**7.14%**	**4.70%**	**0.00%**	8.33%	0.00%	**-5.57%**	0.00%	**2.96%**
		mondial16.db	**-5.55%**		**0.00%**		**-33.35%**		-13.34%	
		mondial32.db	12.50%		25.00%		16.65%		22.22%	
QP3.2	FALSE	mondial1.db	-9.99%	-18.29%	**0.00%**	**0.00%**	0.00%	-6.25%	0.00%	**-3.57%**
		mondial2.db	-27.78%		**0.00%**		0.00%		0.00%	
		mondial3.db	-15.37%		**0.00%**		-24.98%		-14.27%	
		mondial4.db	-20.00%		**0.00%**		0.00%		0.00%	
		mondial8.db	-6.66%	**-3.96%**	**0.00%**	-6.67%	0.00%	**11.92%**	0.00%	**-0.93%**
		mondial16.db	**5.88%**		**0.00%**		**50.04%**		15.39%	
		mondial32.db	-11.11%		-20.00%		-14.27%		-18.18%	

Appendix C

EXPERIMENTAL CONTEXT			EXPERIMENTAL EVALUATION							
EXECUTION INFO			AVERAGE QUERY EXECUTION REAL TIME PERFORMANCE (GROSS MEASURE)				AVERAGE QUERY ‘EFFECTIVE’ CPU EXECUTION TIME PERFORMANCE (NET MEASURE)			
query_pair_index	optimised	database_instance	ABSOLUTE REAL TIME (GROSS)		ABS REAL TIME DIFF	ABS REAL TIME PERCENT DIFF	ABSOLUTE EFFECTIVE CPU TIME (NET)		ABS EFF. CPU TIME DIFF	ABS EFF. CPU TIME PERCENT DIFF
QP1.1	TRUE	mondial1.db mondial2.db mondial3.db mondial4.db	0.005083	0.005042	0.001042	20.66%	0.003834	0.003750	0.000861	22.96%
		mondial8.db mondial16.db mondial32.db	0.005000				0.003667			
QP1.2	FALSE	mondial1.db mondial2.db mondial3.db mondial4.db	0.005167	0.006083			0.003667	0.004611		
		mondial8.db mondial16.db mondial32.db	0.007000				0.005556			
QP2.1	TRUE	mondial1.db mondial2.db mondial3.db mondial4.db	0.008750	0.015764	0.324055	2055.67%	0.007500	0.014250	0.314917	2209.91%
		mondial8.db mondial16.db mondial32.db	0.022778				0.021000			
QP2.2	FALSE	mondial1.db mondial2.db mondial3.db mondial4.db	0.019750	0.339819			0.018000	0.329167		
		mondial8.db mondial16.db mondial32.db	0.659889				0.640333			
QP3.1	TRUE	mondial1.db mondial2.db mondial3.db mondial4.db	0.003750	0.004986	0.000625	12.53%	0.002083	0.003430	0.000153	**4.45%**
		mondial8.db mondial16.db mondial32.db	0.006222				0.004778			
QP3.2	FALSE	mondial1.db mondial2.db mondial3.db mondial4.db	0.004667	0.005611			0.002167	0.003583		
		mondial8.db mondial16.db mondial32.db	0.006556				0.005000			

EXPERIMENTAL CONTEXT KEY

EXECUTION INFO	
TOTAL RELATION	database_instance
	BG COLOUR & GROWTH
x1	Linear
x2	
x3	
x4	
x8	**Exponential**
x16	
x32	

EXECUTION INFO COLUMN		
QUERY PAIR	optimised	
	TRUE	*FALSE*
BG COLOUR	🟩	🟥

EXPERIMENTAL RESULTS & ANALYSIS KEY

MEASURE AVERAGE			
DELTA TIME	QUERY RESULT		Threshold
	Performant	*Inexpedient*	
Equal	🟦	🟦	0.00%
Negligible	🟨	🟨	Δv <= +/- 8%
Considerable	🟩	🟧	Δv > +/- 8%

MEASURE ONE, TWO, THREE		
QUERY PAIR	QUERY TYPE	
	Optimised	*Unoptimised*
BG COLOUR	⬜	⬜

XPECTED EXPERIMENTAL RESULTS ARE BOLD AND UNDERLI

DB_BLOCK := */big{ DB_INSTANCE_{1,2,3,4}, DB_INSTANCE_{8,16,32} \big}*

MEASURE AVERA := *(time_measure_{2} + time_measure_{3} + time_measure_{4}) / 3*

MEASURE COMBIN := *avg_user_time + avg_sys_time*

REL DELTA := *((value_{2} - value_{1}) / value_{1})*

percent_change := *REL_DELTA * 100*

domain_average := *avg(percent_change) | \exists x \in DB_BLOCK_{x} s.t. x <- 1 ∨ 2*

ABS Real (GROS := *avg(MEASURE_AVERAGE_{real-time}) | \forall measures_real_time \in DB_BLOCK_{x} s.t. x <- 1 ∨ 2 ∨ (1 ∧ 2))*

ABS REAL DIFF := *|| ABS Real (GROSS) \forall x \in DB_BLOCK ||*

ABS REAL PERCENT := REL_DELTA (ABS_REAL_GROSS_{2}, ABS_REAL_GROSS_{1}) * 100

ABS CPU NET := *avg(MEASURE_AVERAGE_{user_time+sys_time}) | \forall x \in DB_BLOCK_{x} s.t. x <- 1 ∨ 2 ∨ (1 ∧ 2))*

ABS SPUN DIFF := *|| ABS CPU TIME (USER_TIME + SYS_TIME) (NET) \forall x \in DB_BLOCK ||*

ABS CPUN PERCENT := REL_DELTA (ABS_CPU_NET_{2}, ABS_CPU_NET_{1}) * 100