Group 13 Final Project Report

Members: Daniel Lindsey, Chase Keller, Long Nguyen Submission Date: December 7th, 2025

Proposed Solution:

Our proposed solution for Phase 1 is to run nested and correlated queries alongside their optimized counterparts to demonstrate the efficiency improvements/differences while using medium sized Crime and Consumer Complaint datasets. This will give us an idea as to how much decorrelating and unnesting queries improves run times on actual real-world data. For Phase 2, we aim to identify potential edge cases that cannot be unnested consistently using current methodology and develop algorithms to more efficiently execute them.

Evaluation Results:

To evaluate the relations between correlated and decorrelated queries, 5 queries were generated per category (correlated nested, uncorrelated nested, unnested/decorrelated). In total, there are 30 queries, 15 per dataset, each located on the repository under the dataset directories.

The following table showcases the 3 categories of queries for the Crime dataset, specifically Query 1. Notably, the correlated query compares c1_vict_age > inner row every time the inner record is called, leading to O(n^2) time complexity. The uncorrelated nested and unnested queries use joins on re-usable averages, which lead to quicker execution times as shown in the execution timetables below.

Correlated Nested	Uncorrelated Nested	Unnested/Decorrelated

`SELECT c1.*`
`FROM crimedata_10 c1`
`WHERE c1.vict_age > (`
`  SELECT AVG(c2.vict_age)`
`  FROM crimedata_10 c2`
`  WHERE c2.area = c1.area`
`);`
	`SELECT c1.*`
	`FROM crimedata_10 c1`
	`JOIN (`
	`  SELECT area, AVG(vict_age) AS avg_age`
	`  FROM crimedata_10`
	`  GROUP BY area`
	`) avg_per_area`
	`  ON c1.area = avg_per_area.area`
	`WHERE c1.vict_age > avg_per_area.avg_age;`
		`WITH area_avg AS (`
		`  SELECT area, AVG(vict_age) AS avg_age`
		`  FROM crimedata_10`
		`  GROUP BY area`
		`)`
		`SELECT c1.*`
		`FROM crimedata_10 c1`
		`JOIN area_avg a ON c1.area = a.area`
		`WHERE c1.vict_age > a.avg_age;`

On the other hand, there are cases such as Query 2 for Crime Dataset where each correlation has a less noticeable effect on execution time. In the table below, the correlated query is bound by c2.area, but the presence of “WHERE EXISTS” reduces the cost of correlation as the comparison checks will stop the first time a matching record is found, rather than comparing all the inner records. This optimization leads to more comparable execution times shown in the timetable below under query 2 crime dataset.

Correlated Nested	Uncorrelated Nested

`SELECT c1.*`
`FROM crimedata_10 c1`
`WHERE EXISTS (`
`  SELECT 1`
`  FROM crimedata_10 c2`
`  WHERE c2.area = c1.area`
`  AND c2.crm_cd_desc = 'THEFT OF IDENTITY'`
`);`
	`SELECT *`
	`FROM crimedata_10`
	`WHERE area IN (`
	`  SELECT DISTINCT area`
	`  FROM crimedata_10`
	`  WHERE crm_cd_desc = 'THEFT OF IDENTITY'`
	`);`

Crime Dataset - Full Dataset (1,000,000+ datapoints)

Exec Time(s)	Query 1	Query 2	Query 3	Query 4	Query 5
Correlated	DNF	3.101	DNF	DNF	2.057
Nested	2.045	2.875	0.464	0.419	0.457
Unnested	1.96	2.869	0.436	0.404	0.385

Complaints Dataset - Full Dataset (3,000,000+ datapoints)

Exec Time(s)	Query 1	Query 2	Query 3	Query 4	Query 5
Correlated	DNF	47.478	DNF	DNF	5.35
Nested	2.927	0.377	6.617	6.296	3.386
Unnested	2.922	0.568	6.658	6.1	3.295

Summary:

For Phase 1, we executed 30 subqueries multiple times to measure their average execution times. We then compared the execution times of correlated, nested, and unnested versions of the subqueries. Overall, we saw that unnested/decorrelated versions of the queries executed faster. These results come with the caveat that the PostgreSQL optimizer will still attempt to optimize the queries anyway potentially skewing the results.

In Phase 2, we examined different permutations of SQL queries/commands such as authorization methods, window functions, set returning functions, and hierarchical queries. We weren't able to delve as deeply into these as we initially wanted to, ultimately only ending up being able to conclude whether they can be unnested/decorrelated if included in a subquery and missed out on developing algorithms to eliminate their limitations.

Addressed Comments:

1.	Information leakage can occur if row level security is enabled and the decorrelation/unnesting pushes the has_row_privileges function so that it executes improperly. This can lead to a user gaining access to information they shouldn't.
2.	The PostgreSQL cost-based optimizer attempts to run queries in the most optimal way possible. It does this in various ways, but for our purposes the most important one is how it determines the join order. In cases where the subquery has multiple levels of nesting with partial or full correlation it may be unable to be able to generate a more efficient query plan.
3.	Please see here for an example of the authorization issue.

Repository Link: https://github.com/Daniel-Lindsey66/Group-13-SubQuery-Optimization-Project.git

Note: Includes extra credit opportunities

1. Real World Data (used in testing query run-times)
2. Related Work Comparison

Member Contribution:

Each member provided equal contributions towards the completion of this project.