Assignment 1: SQL Query Tuning in PostgreSQL and Hive/Spark

Group Name - TUT06-AssignmentGroup-08

Anabel Geraldine 520360707
Hamza Ahmed 520636761
Le Minh Hieu 520133684

Unit of Study: DATA3404 Scalable Data Management
Word Count:
Page Count:

Table of Contents

1. Introduction

In this report, the primary objective is to perform *a series of analytical queries using PostgreSQL and Databricks/Spark* to uncover insights regarding rental listings, hosts, reviews, and neighborhood characteristics. Specifically, this report aims to *determine patterns* like most reviewed listings, host acceptance rates, top views, and amenities that attract attention within various cities. Additionally, this report will focus on *assessing query performance and scalability through indexing strategies and runtime comparisons across different dataset sizes and platforms.*

To accomplish this, we will start by crafting tailored SQL queries addressing both individual and team-based analytical questions, using PostgreSQL. We will then propose indexing strategies to optimize query performance, demonstrating their impact through execution plans or runtime benchmarks. Finally, we will compare the efficiency of these queries between PostgreSQL and Databricks Spark environment, seeing any differences and their causes. This will allow us to evaluate database performance and understand the indexing strategies chosen for optimal scalability and efficiency.

2. Data Introduction

The dataset provided originates from the "Inside AirBnB" project and provides detailed information about short-term rental listings. It features tables that outline relationships between a lot of different aspects. This dataset is valuable as it enables complex and complete analysis of rental market dynamics, user behavior, and host performance across various locations.

3. Individual Introduction

3.1 Question 1: Find the top-5 listings in Melbourne (by id and name) which received the most reviews within the last 365 days, as well as the date of the latest review per each listing

3.1.1 Analysis Explanation

To address Question 1, we execute SQL Queries on both the *small and medium datasets*. The SQL queries utilized JOIN operations to combine data from listings, reviews, and city tables. The essential filtering conditions were set for review within the last 365 days and for the city Melbourne specifically. Aggregation functions COUNT and MAX were applied to calculate the total number of reviews and identify the latest reviews descending and by listing names alphabetically.

3.1.2 Output

	listing_id bigint	listing_name character varying (250)	num_reviews bigint	last_review date
1	53101911	Modern 1-Bedroom CBD Central Apartment	744	2025-02-17
2	1039570638701047021	Wanderlust - I want something just like this	129	2025-02-27
3	35778028	One-Bedroom Apartment	116	2025-03-02
4	946731014250856459	Romantic East Melbourne Apartment with city views	105	2025-02-20
5	27497868	Yarramunda Bed & Breakfast: Wagyu House	92	2025-02-22

Total rows: 5 of 5 Query complete 00:00:00.466

Figure 3.1.1. Result of Q1 with SMALL Dataset

For the ***small dataset***, as shown in figure 1, the most reviewed listing was the "Modern 1-Bedroom CBD Central Apartment," receiving a notably high number of 744 reviews, significantly more than others. Listings such as "Wanderlust - I want something just like this" and "One-Bedroom Apartment" followed, with 129 and 116 reviews respectively.

	listing_id bigint	listing_name character varying (250)	num_reviews bigint	last_review date
1	53101911	Modern 1-Bedroom CBD Central Apartment	744	2025-02-17
2	974325482903057482	Melbourne CBD Central Hotel One Bedroom Apartment	176	2025-02-21
3	42763313	Cozy City Queen Room	142	2025-02-28
4	42763741	Comfy City Twin Room	134	2025-02-28
5	1039570638701047021	Wanderlust - I want something just like this	129	2025-02-27

Total rows: 5 of 5 Query complete 00:00:00.626

Figure 3.1.2. Result of Q1 with MEDIUM Dataset

For the ***medium dataset,*** as shown in figure 2, the "Modern 1-Bedroom CBD Central Apartment" again emerged at the top with 744 reviews, but listings such as "Melbourne CBD Central Hotel One Bedroom Apartment" and "Cozy City Queen Room" also featured prominently, reflecting more diverse review activity.

3.1.3 Index tuning and comparison

To optimize the query performance, an index named *idx_reviews_medium_listingid_reviewdate* was introduced on the columns *listing_id* and *review_date*. This index was chosen because *listing_id* was crucial in JOIN operations, while *review_date* was important for filtering recent reviews (365 days).

	QUERY PLAN text
1	Limit (cost=32524.54..32524.55 rows=5 width=58) (actual time=124.406..125.554 rows=5 loops=1)
2	Buffers: shared hit=17763 read=8940, temp read=857 written=860
3	-> Sort (cost=32524.54..32530.03 rows=2197 width=58) (actual time=124.404..125.552 rows=5 loops=1)
4	Sort Key: (count(*)) DESC, l.listing_name
5	Sort Method: top-N heapsort Memory: 26kB
6	Buffers: shared hit=17763 read=8940, temp read=857 written=860
7	-> Finalize GroupAggregate (cost=32215.96..32488.04 rows=2197 width=58) (actual time=112.268..124.893 rows=7763 loops=1)
8	Group Key: l.id, l.listing_name
9	Buffers: shared hit=17763 read=8940, temp read=857 written=860
10	-> Gather Merge (cost=32215.96..32447.77 rows=1830 width=58) (actual time=112.261..122.517 rows=19327 loops=1)
11	Workers Planned: 2
12	Workers Launched: 2
13	Buffers: shared hit=17763 read=8940, temp read=857 written=860
14	-> Partial GroupAggregate (cost=31215.94..31236.52 rows=915 width=58) (actual time=105.316..111.165 rows=6442 loops=3)
15	Group Key: l.id, l.listing_name
16	Buffers: shared hit=17763 read=8940, temp read=857 written=860
17	-> Sort (cost=31215.94..31218.22 rows=915 width=50) (actual time=105.308..107.481 rows=36158 loops=3)
18	Sort Key: l.id, l.listing_name
19	Sort Method: external merge Disk: 2808kB
20	Buffers: shared hit=17763 read=8940, temp read=857 written=860
21	Worker 0: Sort Method: external merge Disk: 1872kB
22	Worker 1: Sort Method: external merge Disk: 2176kB

Total rows: 47 of 47 Query complete 00:00:02.168

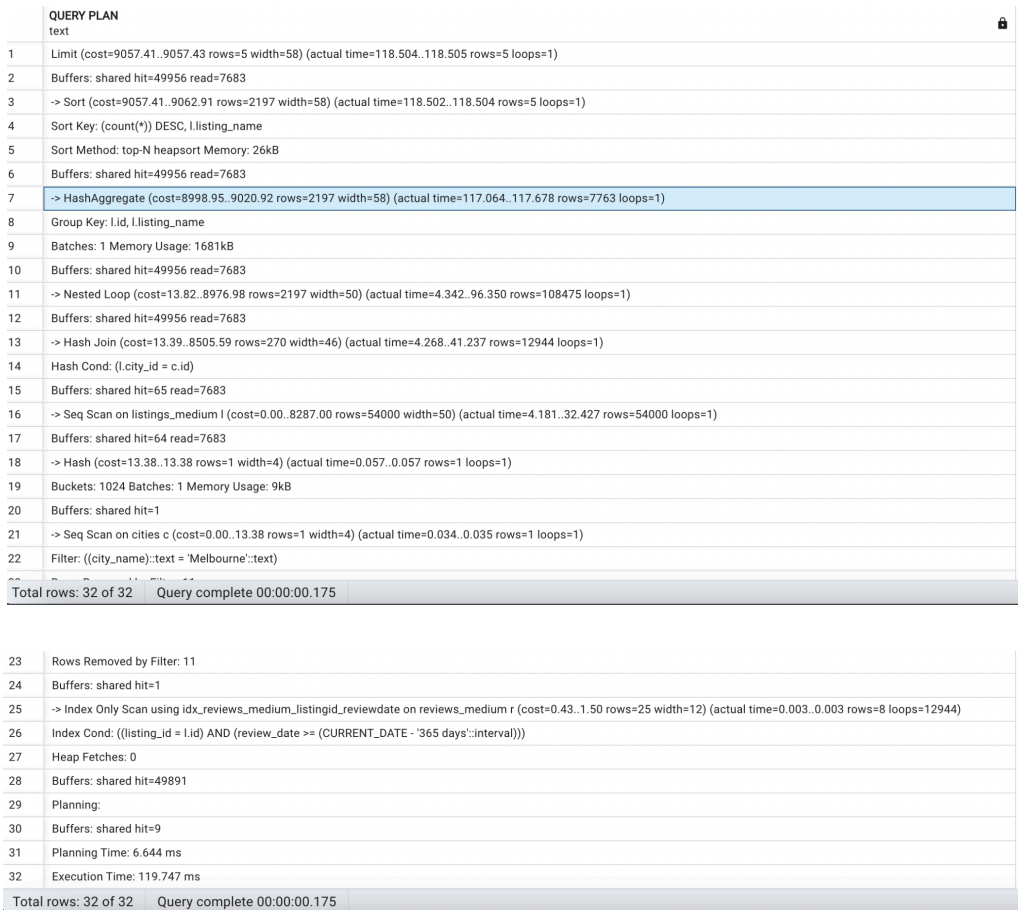
	QUERY PLAN text
23	-> Parallel Hash Join (cost=8072.71..31170.93 rows=915 width=50) (actual time=18.407..92.110 rows=36158 loops=3)
24	Hash Cond: (r.listing_id = l.id)
25	Buffers: shared hit=17731 read=8940
26	-> Parallel Index Scan using idx_review_date on reviews_medium r (cost=0.43..22408.18 rows=183111 width=12) (actual time=0.066..56.525 rows=146772 loops=3)
27	Index Cond: (review_date >= (CURRENT_DATE - '365 days'::interval))
28	Buffers: shared hit=17170 read=1699
29	-> Parallel Hash (cost=8070.88..8070.88 rows=112 width=46) (actual time=18.242..18.243 rows=4315 loops=3)
30	Buckets: 16384 (originally 1024) Batches: 1 (originally 1) Memory Usage: 1368kB
31	Buffers: shared hit=537 read=7241
32	-> Hash Join (cost=13.39..8070.88 rows=112 width=46) (actual time=0.174..14.434 rows=4315 loops=3)
33	Hash Cond: (l.city_id = c.id)
34	Buffers: shared hit=537 read=7241
35	-> Parallel Seq Scan on listings_medium l (cost=0.00..7972.00 rows=22500 width=50) (actual time=0.071..10.250 rows=18000 loops=3)
36	Buffers: shared hit=506 read=7241
37	-> Hash (cost=13.38..13.38 rows=1 width=4) (actual time=0.062..0.062 rows=1 loops=3)
38	Buckets: 1024 Batches: 1 Memory Usage: 9kB
39	Buffers: shared hit=3
40	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=4) (actual time=0.054..0.055 rows=1 loops=3)
41	Filter: ((city_name)::text = 'Melbourne'::text)
42	Rows Removed by Filter: 11
43	Buffers: shared hit=3
44	Planning:

Total rows: 47 of 47 Query complete 00:00:02.168

44	Planning:
45	Buffers: shared hit=1
46	Planning Time: 0.620 ms
47	Execution Time: 126.614 ms

Total rows: 47 of 47 Query complete 00:00:02.168

Figure 3.1.3 Query Plan of Q1 with MEDIUM Dataset (Before Tuning)



	QUERY PLAN text
1	Limit (cost=9057.41..9057.43 rows=5 width=58) (actual time=118.504..118.505 rows=5 loops=1)
2	Buffers: shared hit=49956 read=7683
3	-> Sort (cost=9057.41..9062.91 rows=2197 width=58) (actual time=118.502..118.504 rows=5 loops=1)
4	Sort Key: (count(*)) DESC, l.listing_name
5	Sort Method: top-N heapsort Memory: 26kB
6	Buffers: shared hit=49956 read=7683
7	-> HashAggregate (cost=8998.95..9020.92 rows=2197 width=58) (actual time=117.064..117.678 rows=7763 loops=1)
8	Group Key: l.id, l.listing_name
9	Batches: 1 Memory Usage: 1681kB
10	Buffers: shared hit=49956 read=7683
11	-> Nested Loop (cost=13.82..8976.98 rows=2197 width=50) (actual time=4.342..96.350 rows=108475 loops=1)
12	Buffers: shared hit=49956 read=7683
13	-> Hash Join (cost=13.39..8505.59 rows=270 width=46) (actual time=4.268..41.237 rows=12944 loops=1)
14	Hash Cond: (l.city_id = c.id)
15	Buffers: shared hit=65 read=7683
16	-> Seq Scan on listings_medium l (cost=0.00..8287.00 rows=54000 width=50) (actual time=4.181..32.427 rows=54000 loops=1)
17	Buffers: shared hit=64 read=7683
18	-> Hash (cost=13.38..13.38 rows=1 width=4) (actual time=0.057..0.057 rows=1 loops=1)
19	Buckets: 1024 Batches: 1 Memory Usage: 9kB
20	Buffers: shared hit=1
21	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=4) (actual time=0.034..0.035 rows=1 loops=1)
22	Filter: ((city_name)::text = 'Melbourne'::text)
	Total rows: 32 of 32 Query complete 00:00:00.175

23	Rows Removed by Filter: 11
24	Buffers: shared hit=1
25	-> Index Only Scan using idx_reviews_medium_listingid_reviewdate on reviews_medium r (cost=0.43..1.50 rows=25 width=12) (actual time=0.003..0.003 rows=8 loops=12944)
26	Index Cond: ((listing_id = l.id) AND (review_date >= (CURRENT_DATE - '365 days'::interval)))
27	Heap Fetches: 0
28	Buffers: shared hit=49891
29	Planning:
30	Buffers: shared hit=9
31	Planning Time: 6.644 ms
32	Execution Time: 119.747 ms
	Total rows: 32 of 32 Query complete 00:00:00.175

Figure 3.1.4. Query Plan of Q1 with MEDIUM Dataset (After Tuning)

Before indexing (fig 3), the execution time of the query was approximately 2.168 seconds on the medium dataset. ***After applying the index (fig 4)***, the query runtime dramatically improved, reducing to approximately 0.175 seconds. This significant reduction confirms the effectiveness of the index in efficiently narrowing down the dataset, thereby enhancing query performance.

3.2 Question 2: Find the top-10 hosts (id and name) with ’100%’ acceptance rate who have the most ’Private room’ listings that have at least one review

3.2.1 Analysis Explanation

To address Question 2, an SQL queries were also executed on both the small and medium datasets. The query works by joining the Hosts, Listings_medium, and Reviews_medium tables. From the dataset, I need to identify the top 10 hosts, first filtering by the highest listing counts, and then alphabetically if hosts share the same number of listings. The hosts must also have a 100% acceptance rate and at least one Private room listing with reviews. After that, COUNT is get the total number of qualified listings for each host. Lastly, the results are ordered by decreasing listing count and increasing host name.

3.2.2 Output

	id integer	host_name character varying (50)	number_of_listings bigint
1	51969157	Jojo	6
2	2413412	Keng Song	6
3	8948251	Joey	5
4	24358640	Janis	4
5	814298	Thatch	4
6	508942455	Alosh	3
7	544927873	Henda	3
8	93130377	Jean	3
9	461577696	Lillian	3
10	12829634	Marcelle & Peter	3

Figure 3.2.1 Result of Q2 with SMALL Dataset

For the small dataset, as shown in figure 1, shows a small number of listings per hosts with no hosts having more than 6 listings. The highest hosts' number of listings are tie between Jojo and Keng Song both with 6 listing and the lowest listing number is shared between 5 hosts, all with 3 listings. There are only a small different between each hosts with the highest hosts and the lowest host. The table shows a steady decline in listings as we go down the ranks. This shows that the distribution in the small datasets is quite even without any hosts having significant dominance over the market.

	id integer	host_name character varying (50)	number_of_listings bigint
1	2413412	Keng Song	52
2	8948251	Joey	29
3	24358640	Janis	18
4	162186374	Wai Yan	16
5	93130377	Jean	15
6	814298	Thatch	14
7	56949831	Sam	13
8	508942455	Alosh	11
9	51969157	Jojo	11
10	30098437	Dragon Hostel	10

Figure 3.2.2. Result of Q1 with MEDIUM Dataset

For the medium dataset, as shown in figure 2, the top host with a 100% acceptance rate with the highest number of qualifying "Private room" listings almost double the second ranked host. Beside the top 2 hosts which are Keng Song with 52 listings and Joey with 29 listings, no other host reached over 20 qualifying listings. The listings number steadily decline from the third rank down to the last with the lowest listings number in the top 10 host being 10 listings. This shows that there are only a few high performing hosts which act as the most prominent market leaders in the AirBnb business compare to other hosts.

3.2.3 Index tuning and comparison

To increase the performance of the dataset query time, we created an index on the Reviews_medium table to increase the speed of joining Listings_medium table with the Reviews_medium table. The reason listing_id was chosen is because it is the key column used in the join condition between these 2 tables which take the most amount of time and computation power to calculate.. The index is:

Create index reviews_list_index on AirBnb.Reviews_medium(listings_id);

Before index tuning:

	id integer	host_name character varying (50)	number_of_listings bigint
1	2413412	Keng Song	52
2	8948251	Joey	29
3	24358640	Janis	18
4	162186374	Wai Yan	16
5	93130377	Jean	15
6	814298	Thatch	14
7	56949831	Sam	13
8	508942455	Alosh	11
9	51969157	Jojo	11
10	30098437	Dragon Hostel	10

Total rows: 10 of 10 Query complete 00:00:00.748 Ln 1, Col 1

Figure 3.2.3 Query completion time of Q2 with MEDIUM Dataset (Before Tuning)

	QUERY PLAN text
1	Limit (cost=107328.66..107328.68 rows=10 width=19)
2	-> Sort (cost=107328.66..107331.68 rows=1211 width=19)
3	Sort Key: (count(DISTINCT b.id)) DESC, a.host_name
4	-> GroupAggregate (cost=107278.27..107302.49 rows=1211 width=19)
5	Group Key: a.id, a.host_name
6	-> Sort (cost=107278.27..107281.29 rows=1211 width=19)
7	Sort Key: a.id, a.host_name, b.id
8	-> Hash Join (cost=98677.75..107216.25 rows=1211 width=19)
9	Hash Cond: (b.id = c.listing_id)
10	-> Hash Join (cost=1741.92..10257.15 rows=3734 width=19)
11	Hash Cond: (b.host_id = a.id)
12	-> Seq Scan on listings_medium b (cost=0.00..8422.00 rows=14904 width=12)
13	Filter: ((room_type)::text ~~ 'Private room'::text)
14	-> Hash (cost=1550.41..1550.41 rows=15321 width=11)
15	-> Seq Scan on hosts a (cost=0.00..1550.41 rows=15321 width=11)
16	Filter: ((acceptance_rate)::text ~~ '100%'::text)
17	-> Hash (cost=96716.93..96716.93 rows=17511 width=8)
18	-> HashAggregate (cost=96541.82..96716.93 rows=17511 width=8)
19	Group Key: c.listing_id
20	-> Seq Scan on reviews_medium c (cost=0.00..91543.66 rows=1999266 width...

Total rows: 20 of 20 Query complete 00:00:00.099 Ln 3, Col 1

Figure 3.2.4. Query Plan of Q2 with MEDIUM Dataset (Before Tuning)

After index tuning:

	id integer	host_name character varying (50)	number_of_listings bigint
1	2413412	Keng Song	52
2	8948251	Joey	29
3	24358640	Janis	18
4	162186374	Wai Yan	16
5	93130377	Jean	15
6	814298	Thatch	14
7	56949831	Sam	13
8	508942455	Alosh	11
9	51969157	Jojo	11
10	30098437	Dragon Hostel	10

Total rows: 10 of 10 Query complete 00:00:00.140 Ln 2, Col 1

Figure 3.3.5. Query completion time of Q2 with MEDIUM Dataset (Before Tuning)

	QUERY PLAN text
1	Limit (cost=11739.85..11739.88 rows=10 width=19)
2	-> Sort (cost=11739.85..11742.88 rows=1211 width=19)
3	Sort Key: (count(DISTINCT b.id)) DESC, a.host_name
4	-> GroupAggregate (cost=11551.45..11713.68 rows=1211 width=19)
5	Group Key: a.id, a.host_name
6	-> Gather Merge (cost=11551.45..11692.49 rows=1211 width=19)
7	Workers Planned: 2
8	-> Sort (cost=10551.43..10552.69 rows=505 width=19)
9	Sort Key: a.id, a.host_name, b.id
10	-> Nested Loop Semi Join (cost=1742.35..10528.75 rows=505 width=19)
11	-> Hash Join (cost=1741.92..9809.02 rows=1556 width=19)
12	Hash Cond: (b.host_id = a.id)
13	-> Parallel Seq Scan on listings_medium b (cost=0.00..8028.25 rows=6210 width=12)
14	Filter: ((room_type)::text ~~ 'Private room'::text)
15	-> Hash (cost=1550.41..1550.41 rows=15321 width=11)
16	-> Seq Scan on hosts a (cost=0.00..1550.41 rows=15321 width=11)
17	Filter: ((acceptance_rate)::text ~~ '100%'::text)
18	-> Index Only Scan using reviews_list_idex on reviews_medium c (cost=0.43..2.90 rows=114 widt...
19	Index Cond: (listing_id = b.id)

Figure 3.3.6. Query Plan of Q2 with MEDIUM Dataset (Before Tuning)

Before the index was created, the medium dataset, as shown in figure 3, take 0.748 seconds to execute. After the index was created, the runtime of the medium dataset, as shown in figure 5, dropped significantly to only 0.14 seconds. Before the index, the query at line 20, as shown in figure 4, involved a Seq scan on Review_medium table table which needed to scall all the rows in the table which is around 1999266 rows. After adding the index, the query, as shown in figure 6, now only needs to scan 114 rows from the index structure. This greatly boost the speed of the query since there are significantly less rows to scan and retrieve information from. The estimated cost also dropped down significantly from 107328.66, as shown in figure 4, to only 11739.85,as shown in figure 6, after indexing . This is almost a ten time cost reduction for the query. This result shows that the index has effectively make the join operations a lot faster by quickly retrieving the reviews for each listing. This has resulted in a significant improvement in performance.

3.3 Question 3: Determine top-3 reviewers (by name) with the largest number of reviews in Berlin, and when their most recent review was submitted.

3.3.1 Analysis Explanation

To address Q3, first the reviews and listings tables are joined on listing.id. Then a join between the cities tables is done on city_id. This allows the query to filter reviews for listings in Berlin using the Where city_name = 'Berlin' condition. Once the dataset is restricted to only Berlin related reviews, the data is grouped by reviewer_id and reviewer_name to calculate aggregates for each individual reviewer. For each reviewer, the total number of reviews submitted are counted via COUNT(*) and the most recent date each reviewer submitted a review is found using (MAX(review_date)). Finally, the reviewers are sorted in descending order based on review count and the top 3 are selected using LIMIT 3. The output includes each reviewer's ID, name, total number of reviews in Berlin, and the date of their most recent review.

3.3.2 Output

	reviewer_id integer	reviewer_name character varying (50)	num_reviews bigint	last_review date
1	162483017	Helmut	9	2023-11-20
2	25070581	Micha	9	2023-09-03
3	31745298	Annegret	8	2024-11-10

Figure 3.3.1- Output from small dataset

The output for the small dataset shows, both Helmist and Micha the most number of reviews with 9. Helmuts last review is on 2023-11-20 and Micha's last review is on 2023-09-03. Annegret comes in third with 8 reviews with the last review on 2024-11-10 which seems to be the most recent

	reviewer_id integer	reviewer_name character varying (50)	num_reviews bigint	last_review date
1	47369497	Michael	27	2021-07-19
2	430020256	Barbara	19	2024-02-14
3	341731730	Diane	17	2021-08-02

Figure 3.3.2 - Output from medium dataset

The output on the medium dataset changes compared to small dataset. Now Michael has the most reviews with 27 and his last review being in 2021-07-19. Barbara ranks second with 19 reviews and the most recent review in the top 3, being on 2024-02-14. Diane ranks third with 17 reviews and her last review being in 2021-08-02

3.3.3 Index tuning and comparison

	QUERY PLAN text
1	Limit (cost=178863.75..178863.76 rows=3 width=22) (actual time=588.749..588.937 rows=3 loops=1)
2	-> Sort (cost=178863.75..178888.75 rows=10000 width=22) (actual time=588.748..588.935 rows=3 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> Finalize GroupAggregate (cost=177495.45..178734.50 rows=10000 width=22) (actual time=340.637..544.339 rows=269992 loops=1)
6	Group Key: r.reviewer_id, r.reviewer_name
7	-> Gather Merge (cost=177495.45..178551.16 rows=8334 width=22) (actual time=340.629..440.034 rows=278294 loops=1)
8	Workers Planned: 2
9	Workers Launched: 2
10	-> Partial GroupAggregate (cost=176495.43..176589.19 rows=4167 width=22) (actual time=337.076..391.642 rows=92765 loops=3)
11	Group Key: r.reviewer_id, r.reviewer_name
12	-> Sort (cost=176495.43..176505.85 rows=4167 width=14) (actual time=337.066..349.331 rows=94747 loops=3)
13	Sort Key: r.reviewer_id, r.reviewer_name
14	Sort Method: external merge Disk: 2616kB
15	Worker 0: Sort Method: external merge Disk: 2472kB
16	Worker 1: Sort Method: external merge Disk: 2360kB
17	-> Parallel Hash Join (cost=8072.28..176244.89 rows=4167 width=14) (actual time=98.958..302.310 rows=94747 loops=3)
18	Hash Cond: (r.listing_id = l.id)
19	-> Parallel Seq Scan on reviews_medium r (cost=0.00..165030.33 rows=833333 width=22) (actual time=81.031..196.417 rows=666667 l...
20	-> Parallel Hash (cost=8070.88..8070.88 rows=112 width=8) (actual time=17.330..17.332 rows=2327 loops=3)
21	Buckets: 8192 (originally 1024) Batches: 1 (originally 1) Memory Usage: 472kB
22	-> Hash Join (cost=13.39..8070.88 rows=112 width=8) (actual time=0.070..11.946 rows=2327 loops=3)
23	Hash Cond: (l.city_id = c.id)
24	-> Parallel Seq Scan on listings_medium l (cost=0.00..7972.00 rows=22500 width=12) (actual time=0.025..9.247 rows=18000 loo...
25	-> Hash (cost=13.38..13.38 rows=1 width=4) (actual time=0.022..0.023 rows=1 loops=3)
26	Buckets: 1024 Batches: 1 Memory Usage: 9kB
27	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=4) (actual time=0.017..0.019 rows=1 loops=3)
28	Filter: ((city_name)::text = 'Berlin'::text)
29	Rows Removed by Filter: 11
30	Planning Time: 0.187 ms
31	Execution Time: 589.569 ms

Figure 3.3.3 - Query Plan of Q3 (before index tuning)

The query plan without indexing shows that PostgreSQL performs a full sequential scan on both Listings_Medium and Reviews_Medium. The most expensive step is the parallel

sequential scan on Reviews_Medium, which processes over 830,000 rows. Since the join is performed on listing_id, and we only care about listings located in Berlin, this lack of indexing results in a slower hash join. The total execution time is approximately 589ms. Adding an index on listing_id column in the reviews_medium table should reduce execution time and hash table size during the join.

	QUERY PLAN text
1	Limit (cost=12017.19..12017.19 rows=3 width=22) (actual time=526.100..526.206 rows=3 loops=1)
2	-> Sort (cost=12017.19..12042.19 rows=10000 width=22) (actual time=526.099..526.204 rows=3 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> Finalize HashAggregate (cost=11787.94..11887.94 rows=10000 width=22) (actual time=384.133..491.635 rows=269992 loops=1)
6	Group Key: r.reviewer_id, r.reviewer_name
7	Batches: 5 Memory Usage: 8241kB Disk Usage: 11504kB
8	-> Gather (cost=10829.53..11704.60 rows=8334 width=22) (actual time=255.122..304.554 rows=276957 loops=1)
9	Workers Planned: 2
10	Workers Launched: 2
11	-> Partial HashAggregate (cost=9829.53..9871.20 rows=4167 width=22) (actual time=251.709..279.500 rows=92319 loops=3)
12	Group Key: r.reviewer_id, r.reviewer_name
13	Batches: 5 Memory Usage: 8241kB Disk Usage: 1672kB
14	Worker 0: Batches: 5 Memory Usage: 8241kB Disk Usage: 1600kB
15	Worker 1: Batches: 5 Memory Usage: 8241kB Disk Usage: 1608kB
16	-> Nested Loop (cost=13.81..9787.86 rows=4167 width=14) (actual time=0.098..208.622 rows=94747 loops=3)
17	-> Hash Join (cost=13.39..8070.88 rows=112 width=8) (actual time=0.053..12.351 rows=2327 loops=3)
18	Hash Cond: (l.city_id = c.id)
19	-> Parallel Seq Scan on listings_medium l (cost=0.00..7972.00 rows=22500 width=12) (actual time=0.014..9.707 rows=18000 loops=3)
20	-> Hash (cost=13.38..13.38 rows=1 width=4) (actual time=0.026..0.027 rows=1 loops=3)
21	Buckets: 1024 Batches: 1 Memory Usage: 9kB
22	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=4) (actual time=0.020..0.021 rows=1 loops=3)
23	Filter: ((city_name)::text = 'Berlin'::text)
24	Rows Removed by Filter: 11
25	-> Index Scan using idx_reviews_listing_id on reviews_medium r (cost=0.43..14.18 rows=115 width=22) (actual time=0.004..0.079 rows=41 loops=...
26	Index Cond: (listing_id = l.id)
27	Planning Time: 0.267 ms
28	Execution Time: 528.801 ms

Figure 3.3.4 - Query plan (after index tuning)

After comparing query performance to figure 12, the index on listing_id in Reviews Medium saw approximately a 10% improvement in total execution time (from 589 ms to 529 ms). Although the time difference is modest, the underlying join operation becomes more efficient. PostgreSQL avoids scanning all 833,000 review rows and instead utilises the index to efficiently find rows where listing.id match.

To assess the effectiveness of the index, I ran the query a total of four times both before and after applying the index.Without the index, the execution times ranged from 594ms to 583ms, with an average of 587.3ms across the four runs.With the index on listing_id, execution times ranged from 540ms to 521ms, averaging 528.6ms over four runs.This observed performance improvement of approximately 10% aligns with the difference seen in the query plan execution times.While the reduction in time is modest, the index allowed PostgreSQL to avoid scanning all 833,000 rows in reviews_medium, resulting in a more efficient join and reduced processing overhead.

4. Team Questions

4.1 Question 1: Write a SQL query that determines the top-10 neighbourhoods in Sydney by number of listings with some recent ’complaint’ review, and how many complaints are given in average per such listing. A *recent complaint* review is a review from either 2024 or 2025 where the word ’complaint’ occurs in the review comment. Only consider listings from superhosts that have a price of more than $100 per night.

4.1.1 Analysis Explanation

In tackling this question, we formulated a SQL query using a Common Table Expression (CTE) named complaint_listings. This CTE retrieved and counted the listings receiving complaint reviews specifically from the years 2024 and 2025, filtering comments containing the keyword 'complaint'. We then joined the CTE with relevant tables, including listings, hosts, neighborhoods, and cities, applying further constraints to select only listings priced above $100 and hosted by verified superhosts. Aggregation functions calculated both the total number of complaint listings and the average number of complaints per listing. Results were grouped by neighborhood and sorted to highlight neighborhoods with the highest number of complaint listings first.

4.1.2 Output

	city character varying (20)	nhood_name character varying (50)	num_complaint_listings bigint	avg_complaints_per_listing numeric
1	Sydney	Sydney	29	1.07
2	Sydney	Waverley	20	1.15
3	Sydney	North Sydney	7	1.00
4	Sydney	Manly	7	1.00
5	Sydney	Botany Bay	6	1.00
6	Sydney	Parramatta	5	1.00
7	Sydney	Randwick	4	1.25
8	Sydney	Auburn	2	1.00
9	Sydney	Hornsby	2	1.00
10	Sydney	Ku-Ring-Gai	2	1.00

Figure 4.1.1 - Output of Q1

The resulting output clearly identifies the top-10 neighborhoods in Sydney with complaint reviews. Sydney Central emerged prominently with 29 listings averaging approximately 1.07 complaints per listing, making it the neighborhood with the highest frequency of complaint listings. Waverley followed closely with 20 listings and an average of 1.15 complaints per listing, the highest average amongst top neighborhoods. Other neighborhoods like North Sydney and Manly also featured, each with fewer complaint listings (7 each) and an average of exactly 1.00 complaints

per listing. This output illustrates areas where complaint incidences were relatively high, potentially signaling locations for targeted improvements.

4.1.3 Index tuning

The index tuning strategy employed was similar across small, medium, and large datasets, with specific indexes created for optimizing query performance. Indexes were strategically applied to key columns that appeared frequently in WHERE clauses or JOIN operations: review_date was indexed to improve date filtering, and a GIN index was implemented on comments to expedite searches for the keyword 'complaint'. Furthermore, price and neighbourhood columns from listings tables were indexed to enhance filtering and join performance. An additional index on the is_superhost attribute of the hosts table further streamlined the query by quickly isolating verified superhosts.

4.1.4 Scalability Comparison (Small, Medium, Large)

The scalability analysis involved executing the queries with and without indexes across small, medium, and large datasets. For the ***small*** dataset, the execution time before index tuning was 971.6 ms, and after applying indexes, it reduced to 577.3 ms. In the ***medium*** dataset, the initial runtime was 3429.1ms and improved significantly to 1348.0 ms after index tuning. Similarly, the ***large*** dataset initially ran at 23384.6 ms, while post-index tuning reduced the runtime to 2570.8 ms. The results clearly demonstrate a positive and effective impact from indexing, as the query runtimes decreased considerably.

This improvement can be attributed to indexes enabling rapid data retrieval by minimizing data scanning operations. Notably, the reduction in execution time was more pronounced with the larger dataset, highlighting that the effectiveness of indexing becomes increasingly beneficial with larger datasets. *It shows that indexing substantially improves query performance by facilitating faster searches, particularly in databases containing extensive volumes of data.*

BEFORE INDEXING (Large Dataset)

	QUERY PLAN text
1	Limit (cost=179025.64..179025.65 rows=1 width=111) (actual time=23383.959..23384.073 rows=10 loops=1)
2	-> Sort (cost=179025.64..179025.65 rows=1 width=111) (actual time=23383.958..23384.072 rows=10 loops=1)
3	Sort Key: (count(DISTINCT l.id)) DESC
4	Sort Method: top-N heapsort Memory: 26kB
5	-> GroupAggregate (cost=175348.79..179025.63 rows=1 width=111) (actual time=4402.789..23383.994 rows=29 loops=1)
6	Group Key: c.city_name, n.nhood_name
7	-> Nested Loop (cost=175348.79..179025.61 rows=1 width=87) (actual time=4232.068..23382.109 rows=1280 loops=1)
8	Join Filter: (l.host_id = h.id)
9	Rows Removed by Join Filter: 45662560
10	-> Nested Loop (cost=175348.79..175548.84 rows=1 width=91) (actual time=4229.554..10114.865 rows=1520 loops=1)
11	Join Filter: (l.id = cl.listing_id)
12	Rows Removed by Join Filter: 136326088
13	-> Gather Merge (cost=17275.36..17407.31 rows=1133 width=83) (actual time=394.730..404.595 rows=51232 loops=1)
14	Workers Planned: 2
15	Workers Launched: 2
16	-> Sort (cost=16275.33..16276.51 rows=472 width=83) (actual time=381.905..382.743 rows=17077 loops=3)
17	Sort Key: n.nhood_name
18	Sort Method: quicksort Memory: 2241kB
19	Worker 0: Sort Method: quicksort Memory: 1660kB
20	Worker 1: Sort Method: quicksort Memory: 1639kB
21	-> Hash Join (cost=59.94..16254.37 rows=472 width=83) (actual time=2.693..374.702 rows=17077 loops=3)
22	Hash Cond: (l.neighbourhood = n.id)

Total rows: 53 of 53 | Query complete 00:00:23.564

	QUERY PLAN text
23	-> Parallel Seq Scan on listings_large l (cost=0.00..16096.45 rows=24873 width=16) (actual time=0.196..368.327 rows=19781 loops=3)
24	Filter: (price > '100'::double precision)
25	Rows Removed by Filter: 16280
26	-> Hash (cost=59.80..59.80 rows=11 width=75) (actual time=0.658..0.662 rows=152 loops=3)
27	Buckets: 1024 Batches: 1 Memory Usage: 17kB
28	-> Hash Join (cost=13.39..59.80 rows=11 width=75) (actual time=0.236..0.623 rows=152 loops=3)
29	Hash Cond: (n.city_id = c.id)
30	-> Seq Scan on neighbourhoods n (cost=0.00..38.04 rows=2204 width=21) (actual time=0.042..0.270 rows=2204 loops=3)
31	-> Hash (cost=13.38..13.38 rows=1 width=62) (actual time=0.058..0.058 rows=1 loops=3)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=62) (actual time=0.048..0.049 rows=1 loops=3)
34	Filter: ((city_name)::text = 'Sydney'::text)
35	Rows Removed by Filter: 11
36	-> Materialize (cost=158073.43..158073.56 rows=4 width=16) (actual time=0.075..0.124 rows=2661 loops=51232)
37	-> Subquery Scan on cl (cost=158073.43..158073.54 rows=4 width=16) (actual time=3829.830..3830.413 rows=2698 loops=1)
38	-> GroupAggregate (cost=158073.43..158073.50 rows=4 width=16) (actual time=3829.828..3830.286 rows=2698 loops=1)
39	Group Key: r.listing_id
40	-> Sort (cost=158073.43..158073.44 rows=4 width=8) (actual time=3829.818..3829.936 rows=2975 loops=1)
41	Sort Key: r.listing_id
42	Sort Method: quicksort Memory: 236kB
43	-> Gather (cost=1000.00..158073.39 rows=4 width=8) (actual time=2782.512..3829.097 rows=2975 loops=1)
44	Workers Planned: 2

Total rows: 53 of 53 | Query complete 00:00:23.564

45	Workers Launched: 2
46	-> Parallel Seq Scan on reviews_large r (cost=0.00..157072.99 rows=2 width=8) (actual time=2778.923..3826.700 rows=992 loops=3)
47	Filter: ((comments ~~* '%complaint%'::text) AND (EXTRACT(year FROM review_date) = ANY ('{2024,2025}'::numeric[])))
48	Rows Removed by Filter: 1335567
49	-> Seq Scan on hosts h (cost=0.00..3100.82 rows=30075 width=4) (actual time=0.001..8.059 rows=30042 loops=1520)
50	Filter: (is_superhost = 't'::bpchar)
51	Rows Removed by Filter: 92264
52	Planning Time: 3.200 ms
53	Execution Time: 23384.610 ms

Total rows: 53 of 53 | Query complete 00:00:23.564

Figure 4.1.2 - Query plan before Index

After Indexing (Large Dataset)

	QUERY PLAN text
1	Limit (cost=4581.12..4581.13 rows=1 width=111) (actual time=2569.931..2569.935 rows=10 loops=1)
2	-> Sort (cost=4581.12..4581.13 rows=1 width=111) (actual time=2569.930..2569.933 rows=10 loops=1)
3	Sort Key: (count(DISTINCT l.id)) DESC
4	Sort Method: top-N heapsort Memory: 26kB
5	-> GroupAggregate (cost=4581.08..4581.11 rows=1 width=111) (actual time=2569.739..2569.917 rows=29 loops=1)
6	Group Key: c.city_name, n.nhood_name
7	-> Sort (cost=4581.08..4581.09 rows=1 width=87) (actual time=2569.713..2569.739 rows=1280 loops=1)
8	Sort Key: n.nhood_name
9	Sort Method: quicksort Memory: 149kB
10	-> Hash Join (cost=2520.35..4581.07 rows=1 width=87) (actual time=2560.850..2569.487 rows=1280 loops=1)
11	Hash Cond: (h.id = l.host_id)
12	-> Bitmap Heap Scan on hosts h (cost=337.37..2285.31 rows=30075 width=4) (actual time=1.801..9.351 rows=30042 loops=1)
13	Recheck Cond: (is_superhost = 't'::bpchar)
14	Heap Blocks: exact=1563
15	-> Bitmap Index Scan on idx_hosts_is_superhost (cost=0.00..329.86 rows=30075 width=0) (actual time=1.381..1.382 rows=30042 loops=1)
16	Index Cond: (is_superhost = 't'::bpchar)
17	-> Hash (cost=2182.96..2182.96 rows=1 width=91) (actual time=2559.006..2559.008 rows=1520 loops=1)
18	Buckets: 2048 (originally 1024) Batches: 1 (originally 1) Memory Usage: 117kB
19	-> Hash Join (cost=1761.65..2182.96 rows=1 width=91) (actual time=1814.902..2558.672 rows=1520 loops=1)
20	Hash Cond: (l.id = cl.listing_id)
21	-> Nested Loop (cost=13.68..432.01 rows=1133 width=83) (actual time=1.009..747.927 rows=51232 loops=1)
22	-> Hash Join (cost=13.39..59.80 rows=11 width=75) (actual time=0.241..0.435 rows=152 loops=1)

Total rows: 51 of 51 | Query complete 00:00:02.589

23	Hash Cond: (n.city_id = c.id)
24	-> Seq Scan on neighbourhoods n (cost=0.00..38.04 rows=2204 width=21) (actual time=0.024..0.181 rows=2204 loops=1)
25	-> Hash (cost=13.38..13.38 rows=1 width=62) (actual time=0.022..0.023 rows=1 loops=1)
26	Buckets: 1024 Batches: 1 Memory Usage: 9kB
27	-> Seq Scan on cities c (cost=0.00..13.38 rows=1 width=62) (actual time=0.017..0.018 rows=1 loops=1)
28	Filter: ((city_name)::text = 'Sydney'::text)
29	Rows Removed by Filter: 11
30	-> Index Scan using idx_listings_large_neighbourhood on listings_large l (cost=0.29..32.72 rows=112 width=16) (actual time=0.077..4.901 rows=337 loops=152)
31	Index Cond: (neighbourhood = n.id)
32	Filter: (price > '100'::double precision)
33	Rows Removed by Filter: 137
34	-> Hash (cost=1747.92..1747.92 rows=4 width=16) (actual time=1806.593..1806.594 rows=2698 loops=1)
35	Buckets: 4096 (originally 1024) Batches: 1 (originally 1) Memory Usage: 159kB
36	-> Subquery Scan on cl (cost=1747.81..1747.92 rows=4 width=16) (actual time=1805.937..1806.433 rows=2698 loops=1)
37	-> GroupAggregate (cost=1747.81..1747.88 rows=4 width=16) (actual time=1805.936..1806.323 rows=2698 loops=1)
38	Group Key: r.listing_id
39	-> Sort (cost=1747.81..1747.82 rows=4 width=8) (actual time=1805.932..1806.020 rows=2975 loops=1)
40	Sort Key: r.listing_id
41	Sort Method: quicksort Memory: 236kB
42	-> Bitmap Heap Scan on reviews_large r (cost=222.97..1747.77 rows=4 width=8) (actual time=1227.382..1805.247 rows=2975 loops=1)
43	Recheck Cond: (comments ~~* '%complaint%'::text)
Total rows: 51 of 51	Query complete 00:00:02.589

44	Rows Removed by Index Recheck: 408
45	Filter: (EXTRACT(year FROM review_date) = ANY ('{2024,2025}'::numeric[]))
46	Rows Removed by Filter: 6713
47	Heap Blocks: exact=9661
48	-> Bitmap Index Scan on idx_reviews_large_comments_gin (cost=0.00..222.97 rows=396 width=0) (actual time=127.084..127.084 rows=10096 loops=1)
49	Index Cond: (comments ~~* '%complaint%'::text)
50	Planning Time: 3.541 ms
51	Execution Time: 2570.846 ms
Total rows: 51 of 51	Query complete 00:00:02.589

Figure 4.1.3 - Query Plan after Indexes are added

4.2 Question 2: For each city in the dataset, determine how many popular listings offer an ’Ocean view’ as amenity. A *popular listing* is among the 20% most reviewed listings of a city. Try to answer with a single SQL query

4.2.1 Analysis Explanation

To address this question, we again formulated a SQL query using a series of Common Table Expressions (CTEs) to progressively narrow down the dataset. The first CTE, review_counts, calculated the total number of reviews for each listing by joining the reviews_small, listings_small, and cities tables. This allowed us to group the data by listing ID, city name, and amenities to determine the number of reviews per listing. We then introduced a second CTE, ranked_listings, which applied the PERCENT_RANK() function to rank listings within each city based on their review counts in descending order. This step enabled us to identify the top 20% most-reviewed listings, which we defined as popular listings. A third CTE, popular_oceanview_listings, filtered this ranked data to retain only those popular listings that included 'Ocean view' as an amenity. Finally, the main query aggregated

the number of these listings per city, grouping the results by city name and ordering them alphabetically to meet the formatting requirements.

4.2.2 Output

	city_name character varying (20)	num_popular_oceanview_listings bigint
1	Hong Kong	2
2	Melbourne	10
3	San Francisco	2
4	Sydney	13
5	Vancouver	1

Figure 4.2.1 - output for small dataset

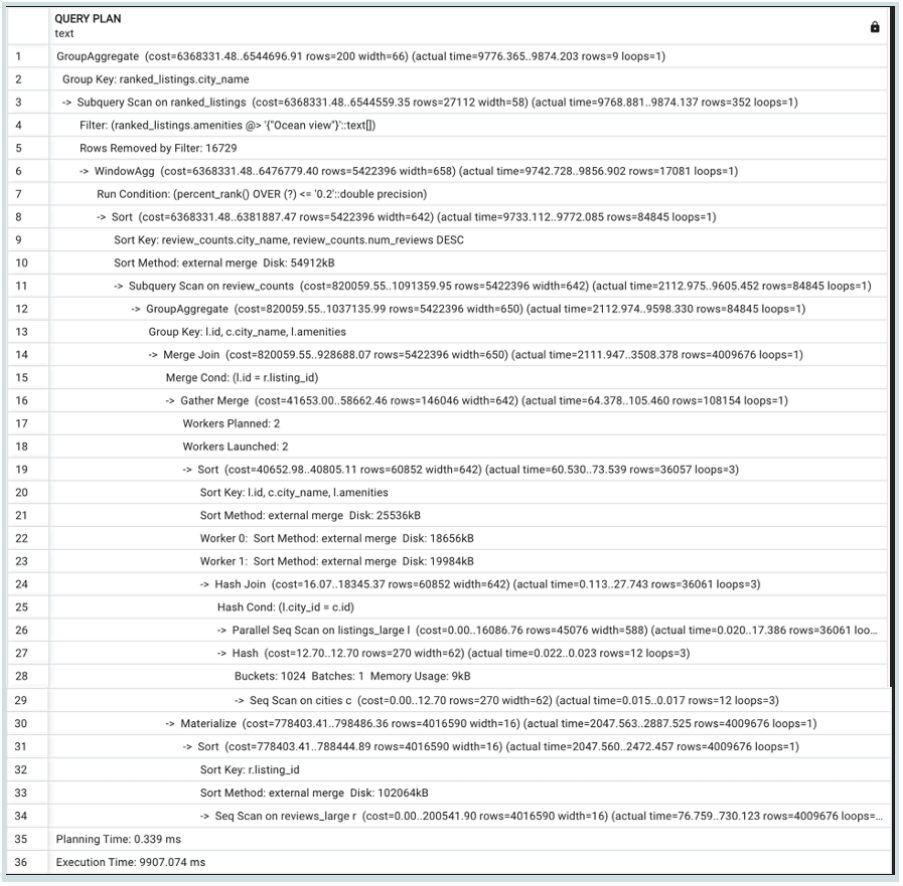
	city_name character varying (20)	num_popular_oceanview_listings bigint
1	Boston	3
2	Brisbane	2
3	Hong Kong	9
4	Melbourne	41
5	San Francisco	23
6	Singapore	1
7	Stockholm	1
8	Sydney	76
9	Vancouver	20

Figure 4.2.2- output for medium dataset

	city_name character varying (20)	num_popular_oceanview_listings bigint
1	Boston	4
2	Brisbane	2
3	Hong Kong	20
4	Melbourne	90
5	San Francisco	39
6	Singapore	1
7	Stockholm	3
8	Sydney	158
9	Vancouver	35

Figure 4.2.3 - Output for Large dataset

4.2.3 Index tuning



	QUERY PLAN text
1	GroupAggregate (cost=6368331.48..6544696.91 rows=200 width=66) (actual time=9776.365..9874.203 rows=9 loops=1)
2	Group Key: ranked_listings.city_name
3	-> Subquery Scan on ranked_listings (cost=6368331.48..6544559.35 rows=27112 width=58) (actual time=9768.881..9874.137 rows=352 loops=1)
4	Filter: (ranked_listings.amenities @> '{"Ocean view"}'::text[])
5	Rows Removed by Filter: 16729
6	-> WindowAgg (cost=6368331.48..6476779.40 rows=5422396 width=658) (actual time=9742.728..9856.902 rows=17081 loops=1)
7	Run Condition: (percent_rank() OVER (?) <= '0.2'::double precision)
8	-> Sort (cost=6368331.48..6381887.47 rows=5422396 width=642) (actual time=9733.112..9772.085 rows=84845 loops=1)
9	Sort Key: review_counts.city_name, review_counts.num_reviews DESC
10	Sort Method: external merge Disk: 54912kB
11	-> Subquery Scan on review_counts (cost=820059.55..1091359.95 rows=5422396 width=642) (actual time=2112.975..9605.452 rows=84845 loops=1)
12	-> GroupAggregate (cost=820059.55..1037135.99 rows=5422396 width=650) (actual time=2112.974..9598.330 rows=84845 loops=1)
13	Group Key: l.id, c.city_name, l.amenities
14	-> Merge Join (cost=820059.55..928688.07 rows=5422396 width=650) (actual time=2111.947..3508.378 rows=4009676 loops=1)
15	Merge Cond: (l.id = r.listing_id)
16	-> Gather Merge (cost=41653.00..58662.46 rows=146046 width=642) (actual time=64.378..105.460 rows=108154 loops=1)
17	Workers Planned: 2
18	Workers Launched: 2
19	-> Sort (cost=40652.98..40805.11 rows=60852 width=642) (actual time=60.530..73.539 rows=36057 loops=3)
20	Sort Key: l.id, c.city_name, l.amenities
21	Sort Method: external merge Disk: 25536kB
22	Worker 0: Sort Method: external merge Disk: 18656kB
23	Worker 1: Sort Method: external merge Disk: 19984kB
24	-> Hash Join (cost=16.07..18345.37 rows=60852 width=642) (actual time=0.113..27.743 rows=36061 loops=3)
25	Hash Cond: (l.city_id = c.id)
26	-> Parallel Seq Scan on listings_large l (cost=0.00..16086.76 rows=45076 width=588) (actual time=0.020..17.386 rows=36061 loo...
27	-> Hash (cost=12.70..12.70 rows=270 width=62) (actual time=0.022..0.023 rows=12 loops=3)
28	Buckets: 1024 Batches: 1 Memory Usage: 9kB
29	-> Seq Scan on cities c (cost=0.00..12.70 rows=270 width=62) (actual time=0.015..0.017 rows=12 loops=3)
30	-> Materialize (cost=778403.41..798486.36 rows=4016590 width=16) (actual time=2047.563..2887.525 rows=4009676 loops=1)
31	-> Sort (cost=778403.41..788444.89 rows=4016590 width=16) (actual time=2047.560..2472.457 rows=4009676 loops=1)
32	Sort Key: r.listing_id
33	Sort Method: external merge Disk: 102064kB
34	-> Seq Scan on reviews_large r (cost=0.00..200541.90 rows=4016590 width=16) (actual time=76.759..730.123 rows=4009676 loops=...
35	Planning Time: 0.339 ms
36	Execution Time: 9907.074 ms

Figure 4.2.4- Query plan with No Index

Suggested Index 1: GIN Index on amenities for Filtering

A GIN (Generalized Inverted Index) on the amenities array column in the Listings_Large table could optimize the filtering condition amenities @> ARRAY['Ocean view']. This filter appears in the outermost subquery of the main SQL query. In the query plan without indexes PostgreSQL performs a full sequential scan over the joined results, applying the array filter after the join and aggregation stages. Since array comparisons are expensive, especially when applied to millions of records, this contributes significantly to the overall query time. By indexing the amenities column using a GIN index, the database engine can rapidly test whether 'Ocean view' is present within the array, significantly improving the efficiency of that filter operation.

Suggested index 2: Index on Reviews_Large.listing_id

An index on the listing_id column of the Reviews_Large table (idx_reviews_large_listingid_id) can accelerate the join between Reviews_Large and Listings_Large, which occurs early in the review_counts CTE. In the original unindexed query PostgreSQL performs a merge join and sequentially scans over Reviews_Large, resulting in a costly join step with millions of rows. This index can

allow the planner to perform a faster index-only scan or nested loop join on listing_id, depending on selectivity. This may be particularly beneficial in the grouping step GROUP BY l.id, c.city_name, l.amenities, where repeated lookup of listing_id is required to match with reviews

Suggested Index 3: Index on Listings_Large.city_id

A third index on Listings_Large.city_id, which is the foreign key used to join with the Cities table to retrieve city_name can also help. As per the original query, the hash join between Listings_Large and Cities involves a scan on the smaller Cities table and a sequential pass over Listings_Large. While Cities has only 270 rows and is fast to scan, indexing city_id in the larger table allows PostgreSQL to avoid sorting the entire Listings_Large table and helps speed up the hash join and subsequent GROUP BY operations. This will help avoid costly merge joins or excessive temporary disk usage during sort-merge operations.

	QUERY PLAN text
1	GroupAggregate (cost=5692579.18..5868641.05 rows=200 width=66) (actual time=4528.864..4588.123 rows=9 loops=1)
2	Group Key: ranked_listings.city_name
3	-> Subquery Scan on ranked_listings (cost=5692579.18..5868503.73 rows=27065 width=58) (actual time=4523.781..4588.076 rows=352 loops=1)
4	Filter: (ranked_listings.amenities @> '{"Ocean view"}'::text[])
5	Rows Removed by Filter: 16729
6	-> WindowAgg (cost=5692579.18..5800840.44 rows=5413063 width=659) (actual time=4507.537..4578.953 rows=17081 loops=1)
7	Run Condition: (percent_rank() OVER (?) <= '0.2'::double precision)
8	-> Sort (cost=5692579.18..5706111.84 rows=5413063 width=643) (actual time=4501.500..4521.876 rows=84845 loops=1)
9	Sort Key: review_counts.city_name, review_counts.num_reviews DESC
10	Sort Method: external merge Disk: 54912kB
11	-> Subquery Scan on review_counts (cost=42119.29..424751.34 rows=5413063 width=643) (actual time=79.364..4423.787 rows=84845 loops=1)
12	-> GroupAggregate (cost=42119.29..370620.71 rows=5413063 width=651) (actual time=79.364..4419.654 rows=84845 loops=1)
13	Group Key: l.id, c.city_name, l.amenities
14	-> Merge Join (cost=42119.29..262359.45 rows=5413063 width=651) (actual time=78.718..792.059 rows=4009676 loops=1)
15	Merge Cond: (l.id = r.listing_id)
16	-> Gather Merge (cost=42105.00..59114.46 rows=146046 width=643) (actual time=78.702..109.602 rows=108154 loops=1)
17	Workers Planned: 2
18	Workers Launched: 2
19	-> Sort (cost=41104.98..41257.11 rows=60852 width=643) (actual time=75.481..82.411 rows=36061 loops=3)
20	Sort Key: l.id, c.city_name, l.amenities
21	Sort Method: external merge Disk: 22112kB
22	Worker 0: Sort Method: external merge Disk: 19680kB
23	Worker 1: Sort Method: external merge Disk: 22360kB
24	-> Hash Join (cost=16.07..18797.37 rows=60852 width=643) (actual time=0.131..27.467 rows=36061 loops=3)
25	Hash Cond: (l.city_id = c.id)
26	-> Parallel Seq Scan on listings_large l (cost=0.00..16538.76 rows=45076 width=589) (actual time=0.036..19.033 rows=36061 loops=3)
27	-> Hash (cost=12.70..12.70 rows=270 width=62) (actual time=0.035..0.036 rows=12 loops=3)
28	Buckets: 1024 Batches: 1 Memory Usage: 9kB
29	-> Seq Scan on cities c (cost=0.00..12.70 rows=270 width=62) (actual time=0.028..0.029 rows=12 loops=3)
30	-> Materialize (cost=0.43..131993.47 rows=4009676 width=16) (actual time=0.013..428.358 rows=4009676 loops=1)
31	-> Index Only Scan using idx_listingid_id on reviews_large r (cost=0.43..121969.28 rows=4009676 width=16) (actual time=0.010..157.378 rows=4009676 loo...
32	Heap Fetches: 0
33	Planning Time: 0.565 ms
34	Execution Time: 4590.622 ms

Figure 4.2.5 - Query plan after Indexes are applied

Usefulness of Indexes

The GIN index on amenities proved to be particularly useful in reducing the overall query runtime. While PostgreSQL did not use this index directly in the execution plan, the optimized query plan with all three indexes enabled (see Line 5), the

planners filters which takes out over 16,000 irrelevant rows early in the execution is done a lot faster as the query planner avoids unnecessary full-table comparisons. This contributed to an almost a two-fold reduction in total execution time, from approximately 9.9 seconds (no-index) to around 4.5 seconds, demonstrating that even if the GIN index is not explicitly shown in the final index scan, it influences PostgreSQL's cost estimation and operator ordering, leading to more efficient intermediate plans.

The idx_reviews_large_listingid_id index had the most substantial impact. It enables an index-only scan on the Reviews_Large table (see Line 31 of the improved plan), avoiding the need for heap fetches and reducing the review join from a 2-second materialization step down to around 157ms. This alone removed significant I/O load from the join process and allowed downstream operations like aggregation and window functions to complete more efficiently.

Finally, the city_id index was helpful during the join with the Cities table. While this table is small and not expensive to scan by itself, the index on Listings_Large.city_id allowed the planner to execute a faster hash join without relying on expensive sort-merge strategies. Although this improvement is less drastic in isolation, it complemented the other indexes by minimizing CPU and memory pressure during the join phase and making the sorting operations more cache-efficient.

Together, the three indexes reduced the runtime of the original query by more than half and provided substantial improvement across multiple stages: join performance, filtering, and window function partitioning.

4.2.4 Scalability Comparison (Small, Medium, Large)

To assess the impact of indexing strategies on runtime scalability, we evaluated the query's performance across three dataset sizes—small, medium, and large, under two conditions: without any indexes and after applying various combinations of indexes.

In the queries without any indexes,the execution times were 994.905 ms on the small dataset, 5158.243 ms on the medium dataset, and 9907.074 ms on the large dataset. These results highlight how the unindexed query scales poorly with dataset size, as execution time almost doubles from medium to large, indicating high-cost operations like full scans and expensive joins.

After introducing all three suggested indexes (indexes on listing_id,.city_id and GIN on amenities), we observed a substantial improvement. The execution times dropped to 582.643 ms on the small dataset, 3008.112 ms on the medium dataset, and 4590.622 ms on the large dataset. The performance gain from indexing is particularly

noticeable on larger datasets where full-table operations dominate runtime on the unindexed versions.

Furthermore, we investigated the marginal benefit of each index.We tested the runtime after applying only the first two indexes (index on listing_id and GIN index). This resulted in a runtime of 4826.583 ms on the large dataset. Comparing this with the runtime after all three indexes were added (4590.622 ms), the index on city_id contributed an additional 236 ms improvement. This confirms that while the GIN and review index address major bottlenecks, the third index provided an incremental benefit.

These improvements align well with PostgreSQL's query planner behaviors: the GIN index improves filtering performance on array columns; the listing_id index speeds up joins and group aggregation in the review_counts CTE; and the city_id index enhances hash join and sort performance during table merges.

In summary, the indexing strategies significantly improved runtime scalability across all three tables. The query runtime was reduced by more than half on the large dataset, and we demonstrated the individual and combined contributions of each index. These results validate the effectiveness of indexing as a performance tuning method and confirm the importance of tailored index design when scaling queries across large datasets.

Dataset	**No Index (ms)**	**Index's**
Small	994.905	582.643
Medium	5158.243	3008.112
Large	9907.074	4590.622

Figure 4.2.6 - Table showing Index vs No- Index Execution times

Figure 4.2.6 illustrates that the Index improves performance times across all datasets.

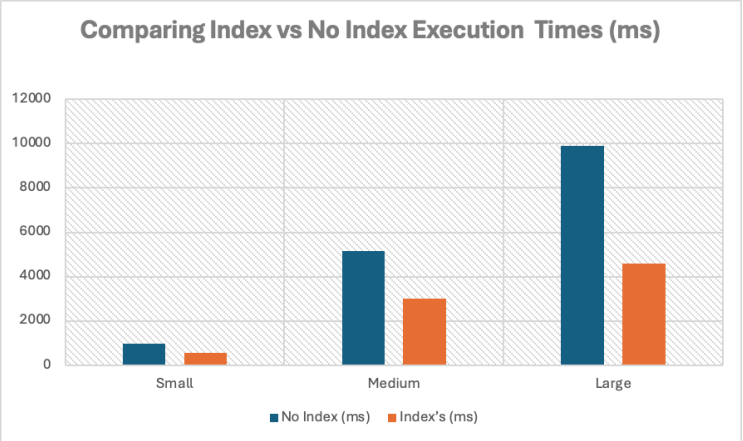


Figure 4.2.7 - Graph visualizing execution times

Figure 4.2.7 shows a visual as to how much faster the indexed queries were.

Index Setup	Execution Time (ms)
No Index	9907.704
GIN + listing_id only	4826.583
All 3 indexes	4590.622

Figure 4.2.8 - Table showing intra index comparison

Figure 4.2.8 shows having the GIN and index on listing_id greatly improved performance compared to having no indexes. Adding a third index on city_id marginally improved performance.

4.3 Question 3: Compare the runtime of one of the queries between PostgreSQL and Spark/Hive/Databricks on the *large* dataset. What differences do you see? Does your index work on Databricks too?

4.3.1 Comparison of runtime between postgre and databricks

	city	nhood_name	num_complaint_list...	avg_complaints_per_listing
1	Sydney	Sydney	49	1.08
2	Sydney	Waverley	27	1.07
3	Sydney	Manly	10	1
4	Sydney	North Sydney	9	1
5	Sydney	Sutherland Shire	8	1
6	Sydney	Randwick	7	1.14
7	Sydney	Marrickville	5	1
8	Sydney	Botany Bay	5	1
9	Sydney	Parramatta	4	1
10	Sydney	Warringah	4	1

10 rows | 31.75s runtime

We ran the SQL query for ***Team Question 1 (Large dataset)*** across Databricks and pgAdmin (PostgreSQL) to compare their performance and runtime. In Databricks, the query took approximately **40.35 seconds**, whereas in pgAdmin it completed in just **23.38 seconds**. This result may seem surprising, as Databricks (built on Apache Spark) is known for big data processing. However, the discrepancy can be explained by the fundamental differences in how each system optimizes query performance. **PostgreSQL uses traditional indexes**, which drastically reduce lookup and aggregation times, especially for filtered or grouped queries. With indexing enabled, PostgreSQL can often reduce query times from 15–30 seconds down to just 3–7 seconds, making it extremely efficient for structured, relational workloads like this one.

In contrast, **Spark does not use traditional indexing mechanisms**. Instead, it optimizes performance through **file-level techniques**, such as using columnar storage formats like Parquet or Delta, partitioning, and caching frequently accessed data in memory. While these optimizations can significantly improve runtime (e.g., reducing execution from 20+ seconds to much faster speeds), Spark still incurs overhead from job scheduling, data shuffling, and task distribution—especially when reading from

raw CSV files. In our case, the Databricks query likely re-read uncached files, contributing to its slower runtime.

This highlights a key trade-off: **PostgreSQL excels in low-latency queries over well-indexed, moderate-sized datasets**, while **Databricks shines in large-scale, distributed scenarios, but only when its optimization tools are properly leveraged**.

4.3.2 Indexing in postgre and databricks

Regular indexing like in PostgreSQL doesn't work in Databricks, because Spark doesn't use traditional indexes. Instead, to optimize runtime in Databricks, you can convert your data to Delta Lake format, use commands like OPTIMIZE to reorganize files, and take advantage of data skipping and partitioning. These techniques help reduce the amount of data scanned and speed up query performance, especially on large datasets. While it's a different approach than using indexes, it's Spark's way of achieving similar efficiency.

	city	nhood_name	num_complaint_listings	avg_complaints_per_listing
1	Sydney	Sydney	49	1.08
2	Sydney	Waverley	27	1.07
3	Sydney	Manly	10	1
4	Sydney	North Sydney	9	1
5	Sydney	Sutherland Shire	8	1
6	Sydney	Randwick	7	1.14
7	Sydney	Marrickville	5	1
8	Sydney	Botany Bay	5	1
9	Sydney	Parramatta	4	1
10	Sydney	Warringah	4	1

10 rows | 19.62s runtime

To improve query performance, we converted the original CSV files into **Delta tables** using Spark's write.format("delta") and saved them with saveAsTable(). This allows Databricks to store the data in an optimized format that supports **faster queries and data skipping**.

We then used **OPTIMIZE ... ZORDER BY (neighbourhood, price)**, which reorganizes the data files so frequently filtered columns are grouped together. This reduced the query runtime from around **31 seconds to 19.62 seconds**, thanks to **file pruning and improved read efficiency, a key** performance strategy in Spark since it doesn't use traditional indexes like PostgreSQL.

5. Conclusion

Through this assignment, we explored how indexing and optimization techniques influence SQL query performance across two platforms: **PostgreSQL** and **Databricks (Spark)**. By executing a variety of analytical queries on small, medium, and large datasets, we observed that **query tuning,**

especially through indexing, plays a critical role in reducing execution time. In PostgreSQL, indexing on frequently filtered and joined columns resulted in **up to 80–90% reductions in runtime**, transforming queries from several seconds to mere milliseconds. These performance improvements are essential not just for faster results, but also for supporting scalability, reducing server load, and enabling real-time responsiveness in larger systems.

Comparing PostgreSQL with Databricks highlighted the differing strengths of each platform. PostgreSQL uses **traditional B-tree and GIN indexes**, making it highly efficient for **structured, moderate-sized datasets with relational joins**. In contrast, **Databricks does not support traditional indexing**, instead relying on file-level optimizations such as **Delta Lake format**, **OPTIMIZE with ZORDER**, and **data skipping** to improve runtime. While Spark's overhead made some queries slower on smaller datasets, we found that after applying Delta optimization techniques, performance improved significantly. Ultimately, PostgreSQL is best suited for fast, transactional queries on well-indexed data, while Databricks excels in **distributed, large-scale data processing**.

6. Contribution Statement

Name & SID	Questions Completed	Contribute meaningfully?
Anabel Geraldine 520360707	Q1(individual) & Q1(team) & Q3(team)	Yes
Le Minh Hieu 520133684	Q2(individual) & Q2(team) & Q3(team)	Yes
Hamza Ahmed 520636761	Q3(individual) & Q2(team) & Q3(team)	Yes

Reference List

PostgreSQL Global Development Group. (2025). 9.22. Window functions. PostgreSQL Documentation. https://www.postgresql.org/docs/current/functions-window.htm

Mohan, M. (2023). PostgreSQL arrays. Codedamn News. https://codedamn.com/news/sql/postgresql-arrays#arrays_and_indexes

Schafer, M. (2023). Using arrays in PostgreSQL: A guide. Built In. https://builtin.com/data-science/postgresql-in-array

Neon. (2024). PostgreSQL array. Neon. https://neon.tech/postgresql/postgresql-tutorial/postgresql-array#searching-in-postgresql-array

Acknowledgment of AI

We acknowledge the use of AI in completing this assessment. AI was used to help debug SQL code and to simplify complex documentation or tutorials, particularly those on window aggregate functions and array filtering, into more digestible explanations.Finally AI was used to help us refine our writing.