<u>Question Q1.1 View A (1 Mark):</u>

- Confirming most popular genre (Works); determined by no. of times item is borrowed or held.
- Whereby, 'borrowed' and 'held' are 'Loan' and 'Hold' respectively in event_type.

SQL Query For Top 5 Most Popular Genres:

CREATE VIEW V_POPULAR_GENRES AS
SELECT genre, COUNT(*) AS total
FROM Works W
JOIN Items I ON W.isbn = I.isbn
JOIN Events E ON I.item_id = E.item_id
WHERE E.event_type IN ('Loan', 'Hold')
GROUP BY genre
ORDER BY total DESC
LIMIT 5;

Reporting Results Using Query SELECT * FROM V_POPULAR_GENRES;

```
a2=# SELECT * FROM V_POPULAR_GENRES;
      genre      | total
-----------------+-------
 Science Fiction | 20297
 Fantasy         | 19568
 Romance         | 18994
 Young Adult     | 18883
 Thriller        | 17428
(5 rows)

Time: 63.055 ms
```

Reporting Query Plan: EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;

```
a2=# EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;
                                                            QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=6801.62..6801.64 rows=5 width=18) (actual time=57.901..71.352 rows=5 loops=1)
   ->  Sort  (cost=6801.62..6801.65 rows=9 width=18) (actual time=57.900..71.350 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=6800.31..6801.48 rows=9 width=18) (actual time=57.891..71.346 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=6800.31..6801.35 rows=9 width=18) (actual time=57.886..71.338 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=5800.30..5800.32 rows=9 width=18) (actual time=52.450..52.453 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=5800.07..5800.16 rows=9 width=18) (actual time=52.381..52.384 rows=9 loops=2)
                                 Group Key: w.genre
                                 Batches: 1  Memory Usage: 24kB
                                 Worker 0:  Batches: 1  Memory Usage: 24kB
                                 ->  Hash Join  (cost=521.07..5338.87 rows=92240 width=10) (actual time=4.925..43.424 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.57..5061.22 rows=92240 width=14) (actual time=4.568..32.689 rows=78617 loops=2)
                                             Hash Cond: ((e.item_id)::text = (i.item_id)::text)
                                             ->  Parallel Seq Scan on events e  (cost=0.00..4332.45 rows=92240 width=14) (actual time=0.029..15.984 rows=78617 loops=2)
                                                   Filter: ((event_type)::text = ANY ('{Loan,Hold}'::text[]))
                                                   Rows Removed by Filter: 59386
                                             ->  Hash  (cost=282.92..282.92 rows=16292 width=28) (actual time=4.429..4.429 rows=16292 loops=2)
                                                   Buckets: 16384  Batches: 1  Memory Usage: 1083kB
                                                   ->  Seq Scan on items i  (cost=0.00..282.92 rows=16292 width=28) (actual time=0.015..2.229 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.327..0.327 rows=1000 loops=2)
                                             Buckets: 1024  Batches: 1  Memory Usage: 64kB
                                             ->  Seq Scan on works w  (cost=0.00..22.00 rows=1000 width=24) (actual time=0.013..0.206 rows=1000 loops=2)
 Planning Time: 0.852 ms
 Execution Time: 71.811 ms
(32 rows)

Time: 73.603 ms
```

- We see that the cost increased slightly from 6801.62 to 6801.64, which is efficient limiting to the top 5 genres. However, the operation took 13.5ms to return the 5 rows, which, though simple, still required some processing time. We also see that the cost raised slightly from 6801.62 to 6801.65,

which indicates low overhead for sorting 9 rows. However, the sorting took 13.45ms, which is relatively time-consuming.

- Moving on, we notice the first cost jumped from 521.07 to 5338.87, which reflected the significant work required to join the 92,240 rows. Furthermore, the execution time was 43.42ms to process 78,617 rows --> this is arguably typical for hash joins due to the creation of hash tables for row comparison. Whereas the second cost hash join saw an increase of cost from 486.57 to 5061.22 --> this again indicates the resource-intensive nature of joining such large datasets; it took 32.69ms to process 78,617 rows.

- Another important point to mention is the involvement of the parallel seq scan on Events; the cost starts at 0.00 and rose to 4332.45 when scanning the 92,240 rows sequentially (no index used). Here we see the filter removed 59,386 rows, leaving 78,617 rows for any further steps. We see that the initial 0.00 cost reflects the minimal prep time for this first operation.

- Planning Time: The 0.852ms planning time shows that PostgreSQL had efficiently optimised the query --> choosing the best path for joins based on table size, indexes, and data distribution.
- Execution Time: The total execution took 71.811ms which is quite reasonable given the volume of the large dataset and the numerous complex joins.

Question Q1.2 VIEW B (2 Marks):

- Identifying top 5 patrons responsible for paying most charges for their children (Jan - June 2024); charges are when the child loses an item.
- Implementing Hint to use COMMON TABLE EXPRESSIONS (CTE) - these are temporary

Planning Steps:

- Which children incurred charges for "Loss" event_type items during Jan 2024 to June 2024.
- Filter children = looking where age <18, then we find corresponding guardian of the child. Note, guardians can have more than one child.
- Group guardian and sum charges for all children under that guardian's name; order top 5.
- Assume that children must have a guardian (guardian IS NOT NULL)

SQL Query For Top 5 Patrons Paying The Most Charges:

```
CREATE VIEW V_COSTS_INCURRED AS
WITH Child_Losses AS (
    SELECT E.patron_id AS ChildId, E.charge, E.time_stamp
    FROM Events E
    JOIN Patrons P ON E.patron_id = P.patron_id
    WHERE E.event_type = 'Loss'
    AND AGE(P.dob) < INTERVAL '18 years'
    AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
),
GuardianCharge AS (
    SELECT SUM(C.charge) AS TotalCharges, P.guardian
```

```
    FROM Child_Losses C
    JOIN Patrons P ON C.ChildId = P.patron_id
    WHERE P.guardian IS NOT NULL
    GROUP BY P.guardian
)
SELECT GC.guardian, GC.TotalCharges
FROM GuardianCharge GC
ORDER BY GC.TotalCharges DESC
LIMIT 5;
```

Reporting Results Using Query SELECT * FROM V_COSTS_INCURRED;

```
a2=# SELECT * FROM V_COSTS_INCURRED;
 guardian | totalcharges
----------+--------------
      295 |       150113
      282 |       130584
      292 |       111350
      286 |       109564
      291 |        95384
(5 rows)

Time: 26.480 ms
```

Question Q1.3 Materialised View (2 Marks):

(1) Create Materialised View Called MV_COSTS_INCURRED (0 Marks):

```
CREATE MATERIALIZED VIEW MY_COSTS_INCURRED AS
WITH Child_Losses AS (
    SELECT E.patron_id AS ChildId, E.charge, E.time_stamp
    FROM Events E
    JOIN Patrons P ON E.patron_id = P.patron_id
    WHERE E.event_type = 'Loss'
    AND AGE(P.dob) < INTERVAL '18 years'
    AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
),
GuardianCharge AS (
    SELECT SUM(C.charge) AS TotalCharges, P.guardian
    FROM Child_Losses C
    JOIN Patrons P ON C.ChildId = P.patron_id
    WHERE P.guardian IS NOT NULL
    GROUP BY P.guardian
)
SELECT GC.guardian, GC.TotalCharges
FROM GuardianCharge GC
ORDER BY GC.TotalCharges DESC
LIMIT 5;
```

(2) Report, Compare, And Explain 2 Following Queries (2 Marks):

- Query A: SELECT * FROM V_COSTS_INCURRED;
- Query B: SELECT * FROM MV_COSTS_INCURRED;

```
a2=# SELECT * FROM MY_COSTS_INCURRED;
 guardian | totalcharges
----------+--------------
      295 |       150113
      282 |       130584
      292 |       111350
      286 |       109564
      291 |        95384
(5 rows)

Time: 0.276 ms
```

```
a2=# SELECT * FROM V_COSTS_INCURRED;
 guardian | totalcharges
----------+--------------
      295 |       150113
      282 |       130584
      292 |       111350
      286 |       109564
      291 |        95384
(5 rows)

Time: 26.480 ms
```

- At a glance, we notice that Query A has a much slower execution time of 26.480ms (relative to Query B's faster 0.276ms).

- Using EXPLAIN ANALYZE to compare both query plans, Query A involves a parallel sequential scan on the events table, which initially processes 276,005 rows. After applying filters for the time range and event type, we see that 147, 486 rows are removed - leaving 517 rows for further processing. The key performance bottleneck in Query A is the sequential scan, which has to process a large dataset before applying the necessary filters and joins; this contributes to the slower execution time.

- In contrast, Query B is substantially faster since it only performs a simple sequential scan on the precomputed data from the materialised view, processing just 5 rows. This avoids the costly sequential scan, filtering and join operations in Query A, resulting in a much faster query execution.

Question 2 Q2.1 Basic Index (2 Marks):

(1) Create Index IDX_EVENT_ITEM On The Event Type And item_id Fields For The Events Table (0.5 Marks):

CREATE INDEX IDX_EVENT_ITEM ON Events (event_type, item_id);

```
a2=# CREATE INDEX IDX_EVENT_ITEM ON Events (event_type, item_id);
CREATE INDEX
Time: 565.654 ms
```

<u>(2) Re-Run Your Query From Q1.1 Again. Report The Query Plan For The Index And Compare (1.5 Marks):</u>

- Creating the index improves the execution time significantly, reducing it by about 26%
- With index = 53.027ms execution time
- Without index = 71.811ms execution time

```
a2=# EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=6801.62..6801.64 rows=5 width=18) (actual time=57.901..71.352 rows=5 loops=1)
   ->  Sort  (cost=6801.62..6801.65 rows=9 width=18) (actual time=57.900..71.350 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=6800.31..6801.48 rows=9 width=18) (actual time=57.891..71.346 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=6800.31..6801.35 rows=9 width=18) (actual time=57.886..71.338 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=5800.30..5800.32 rows=9 width=18) (actual time=52.450..52.453 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=5800.07..5800.16 rows=9 width=18) (actual time=52.381..52.384 rows=9 loops=2)
                                 Group Key: w.genre
                                 Batches: 1  Memory Usage: 24kB
                                 Worker 0:  Batches: 1  Memory Usage: 24kB
                                 ->  Hash Join  (cost=521.07..5338.87 rows=92240 width=10) (actual time=4.925..43.424 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.57..5061.22 rows=92240 width=14) (actual time=4.568..32.689 rows=78617 loops=2)
                                             Hash Cond: ((e.item_id)::text = (i.item_id)::text)
                                             ->  Parallel Seq Scan on events e  (cost=0.00..4332.45 rows=92240 width=14) (actual time=0.029..15.984 rows=78617 loops=2)
                                                   Filter: ((event_type)::text = ANY ('{Loan,Hold}'::text[]))
                                                   Rows Removed by Filter: 59386
                                             ->  Hash  (cost=282.92..282.92 rows=16292 width=28) (actual time=4.429..4.429 rows=16292 loops=2)
                                                   Buckets: 16384  Batches: 1  Memory Usage: 1083kB
                                                   ->  Seq Scan on items i  (cost=0.00..282.92 rows=16292 width=28) (actual time=0.015..2.229 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.327..0.327 rows=1000 loops=2)
                                             Buckets: 1024  Batches: 1  Memory Usage: 64kB
                                             ->  Seq Scan on works w  (cost=0.00..22.00 rows=1000 width=24) (actual time=0.013..0.206 rows=1000 loops=2)
 Planning Time: 0.852 ms
 Execution Time: 71.811 ms
(32 rows)

Time: 73.603 ms
```

```
a2=# EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=5392.48..5392.49 rows=5 width=18) (actual time=43.050..52.870 rows=5 loops=1)
   ->  Sort  (cost=5392.48..5392.50 rows=9 width=18) (actual time=43.049..52.869 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=5391.17..5392.34 rows=9 width=18) (actual time=43.039..52.864 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=5391.17..5392.20 rows=9 width=18) (actual time=43.034..52.856 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=4391.16..4391.18 rows=9 width=18) (actual time=38.050..38.052 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=4390.93..4391.02 rows=9 width=18) (actual time=38.005..38.009 rows=9 loops=2)
                                 Group Key: w.genre
                                 Batches: 1  Memory Usage: 24kB
                                 Worker 0:  Batches: 1  Memory Usage: 24kB
                                 ->  Hash Join  (cost=521.49..3929.73 rows=92240 width=10) (actual time=4.510..30.059 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.99..3652.07 rows=92240 width=14) (actual time=4.145..20.286 rows=78617 loops=2)
                                             Hash Cond: ((e.item_id)::text = (i.item_id)::text)
                                             ->  Parallel Index Only Scan using idx_event_item on events e  (cost=0.42..2923.31 rows=92240 width=14) (actual time=0.071..6.316 rows=78617 loops=2)
                                                   Index Cond: (event_type = ANY ('{Loan,Hold}'::text[]))
                                                   Heap Fetches: 0
                                             ->  Hash  (cost=282.92..282.92 rows=16292 width=28) (actual time=4.002..4.003 rows=16292 loops=2)
                                                   Buckets: 16384  Batches: 1  Memory Usage: 1083kB
                                                   ->  Seq Scan on items i  (cost=0.00..282.92 rows=16292 width=28) (actual time=0.015..2.243 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.336..0.336 rows=1000 loops=2)
                                             Buckets: 1024  Batches: 1  Memory Usage: 64kB
                                             ->  Seq Scan on works w  (cost=0.00..22.00 rows=1000 width=24) (actual time=0.013..0.209 rows=1000 loops=2)
 Planning Time: 0.869 ms
 Execution Time: 53.027 ms
(32 rows)

Time: 54.539 ms
```

Sequential Scan Vs Index Scan:

- A parallel sequential scan is performed on the Events table with a cost starting at 0.00 and then rising to 4332.45; this scan possess 92, 250 rows, removing 59, 386 rows based on the filter event_type being either Hold or Loan --> Note, the sequential scan is slower since the system needs to scan every row, regardless of whether it matches the filter condition.

- However, with the index, we use a Parallel Index Only Scan, which significantly reduces the cost, starting at 0.4 and ending at 2923.31. The index scan will process the same 92, 240 rows but does so in a more efficient way by directly accessing only the relevant rows (only the rows that meet the filter event_type = 'Loan' or 'Hold'). This leads to a much faster execution time, from 0.071ms to 6.316ms (compared to the sequential scan's 0.029ms to 15.984ms).

- Because of this parallel index only scan, the hash joins also become more efficient in the query

plan --> due to the reduced input from the Events time; the cost and execution times are lower because fewer rows need to be processed. For instance, the first hash join between ITEMS and WORKS, without an index, starts at cost 521.07 to 5338.87, with an execution time of 43.424ms for 78, 617 rows. Whereas with an index, the first hash join between ITEMS AND WORKS starts at cost 521.07 to 3929.73, with an execution time of 30.059ms for 78, 617 rows.

Question Q2.2 Function-Based Index (4 Marks):

(1) Create An Expression To Identify Each Author's Surname (1 Mark):

- Surname from author field = last space-delimited word
- Extracts last word after a space (from author in Works table)
- regexp_substr --> function that extracts substring from input string

SELECT regexp_substr(author, ' (.*)', 1, 1, 'c') AS surname FROM Works;

(2) Report And Explain The Query Plan (1 Mark):

```
a2=# EXPLAIN ANALYZE SELECT regexp_substr(author, ' (.*)', 1, 1, 'c') AS surname FROM Works;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on works  (cost=0.00..24.50 rows=1000 width=32) (actual time=0.019..0.696 rows=1000 loops=1)
 Planning Time: 0.042 ms
 Execution Time: 0.736 ms
(3 rows)

Time: 1.038 ms
```

- The cost starts at 0.00 and raised to 24.50; this low cost is expected namely since the query is just selecting and manipulating the author field without any complex filtering or join operations (this query is lightweight because it is only extracting a part of the varchar field) --> behaving similar to that of a simple SELECT statement.
- The simplicity of the regex extraction as well as its single loop makes the query fast with an execution time of 0.736ms!

(3) Create a Function-Based Index To Potentially Speed Up Queries On This Expression (0.5 Marks):

CREATE INDEX IDX_AUTHOR ON Works (author);

(4) Report The Query Plan With The Function-Based Index In Place. Explain The Differences To The Previous Plan, If Any (1.5 Marks):

```
a2=# EXPLAIN ANALYZE SELECT regexp_substr(author, ' (.*)', 1, 1, 'c') AS surname FROM Works;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on works  (cost=0.00..24.50 rows=1000 width=32) (actual time=0.021..0.702 rows=1000 loops=1)
 Planning Time: 0.058 ms
 Execution Time: 0.747 ms
(3 rows)

Time: 1.074 ms
```

- After creating the index, the query plan still shows a sequential scan with execution time 0.747ms on the same 1000 rows with the cost remaining 0.00 to 24.50. We can deduce that PostgreSQL opted not to use the index - rather, it decided that the Seq Scan was still the most efficient way to retrieve all the rows.

- The planning time slightly increased from 0.042ms to 0.058ms because PostgreSQL's planner considered whether, for the most optimal plan, to use the index or the sequential scan.

Q3 Indexes And Query Planning (4 Marks):

- We will be suppressing the use of several different scan types to compare queries.
- Reset both index and sequential scans back to on before proceeding to another question.
- For this Question, bitmap and index-only scans count as index scans too.
- Report and explain the execution plan for the following two queries (A & B):

Query A: SELECT * FROM Events WHERE event_id < 100;
Query B: SELECT * FROM Events WHERE event_id >= 100;

(1) Condition 1: With Both Index And Sequential Scans Enabled:

SET enable_seqscan = ON;
SET enable_indexscan = ON;
SET enable_bitmapscan = ON;
SET enable_indexonlyscan = ON;

```
a2=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id < 100;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10.24 rows=104 width=39) (actual time=0.004..0.012 rows=99 loops=1)
   Index Cond: (event_id < 100)
 Planning Time: 0.177 ms
 Execution Time: 0.026 ms
(4 rows)

Time: 0.439 ms
```

```
a2_1=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id >= 100;
                                                       QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Seq Scan on events  (cost=0.00..5753.06 rows=275908 width=39) (actual time=0.010..17.340 rows=275906 loops=1)
   Filter: (event_id >= 100)
   Rows Removed by Filter: 99
 Planning Time: 0.089 ms
 Execution Time: 25.405 ms
(5 rows)

Time: 25.779 ms
```

Query A:

The query planner optimises Query A by using an index scan on the primary key, allowing it to significantly reduce execution time. The condition event_id < 100 enables the query to leverage the events_pkey index to efficiently scan only the relevant rows. Consequently, only 99 rows are fetched, minimising the overall effort. This is reflected in the cost range from 0.42 to 10.24, indicating that the system expends minimal resources to retrieve these few rows. Plus, the execution time of 0.026ms further highlights the speed and efficiency gained by using the index scan, making this query incredibly fast!

Query B:

For Query B, the query planner opts for sequential scan rather than using an index --> this is because the condition event_id >= 100 requires scanning a large portion of the table (275,908 rows). In this case, a sequential scan is more efficient than an index scan when it comes to processing such a large number of rows; this results in a longer execution time of 25.405ms compared to Query A.

We see that the cost starts at 0.00 and rises to 5753.06, reflecting the extensive effort required to scan through the large dataset.

(2) Condition 2: With Index Scans Enabled And Sequential Scans Suppressed:

SET enable_seqscan = OFF;
SET enable_indexscan = ON;
SET enable_bitmapscan = ON;
SET enable_indexonlyscan = ON;

```
a2_1=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id < 100;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10.12 rows=97 width=39) (actual time=0.004..0.011 rows=99 loops=1)
   Index Cond: (event_id < 100)
 Planning Time: 0.073 ms
 Execution Time: 0.025 ms
(4 rows)

Time: 0.385 ms
```

```
a2_1=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id >= 100;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10170.81 rows=275908 width=39) (actual time=0.010..25.124 rows=275906 loops=1)
   Index Cond: (event_id >= 100)
 Planning Time: 0.094 ms
 Execution Time: 33.384 ms
(4 rows)

Time: 33.765 ms
```

Query A:

Query A uses an index scan because it is retrieving rows with event_id < 100 --> which only involves 99 rows. Even though sequential scans are disabled, the index on the primary key events_pkey is still optimal due to the small subset of rows being fetched. We see that the cost starts at 0.42 and ends at 10.12, reflecting minimal computational effort.

Query B:

Since sequential scans are disabled, the query is forced to use the index scan on the primary key events_pkey for event_id >= 100. This results in a significantly higher cost and execution time compared to Query A. We see that the cost begins at 0.42 and ends at 10,170.81, as the query processes a substantial 275,908 rows. The issue here is that the index scan, while efficient for small subsets of rows, becomes less optimal for these larger datasets; the overhead of using using the index to locate and retrieve such a large number of rows makes the process slower than a sequential scan. Consequently, this leads to the execution time increasing considerably to 33.384ms.

(3) With Index Scans Suppressed And Sequential Scans Enabled:

SET enable_seqscan = ON;
SET enable_indexscan = OFF;
SET enable_bitmapscan = OFF;
SET enable_indexonlyscan = OFF;

```
a2=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id >= 100;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on events  (cost=0.00..5753.06 rows=275911 width=39) (actual time=0.010..17.236 rows=275906 loops=1)
   Filter: (event_id >= 100)
   Rows Removed by Filter: 99
 Planning Time: 0.081 ms
 Execution Time: 25.330 ms
(5 rows)

Time: 25.701 ms
```

```
a2=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id < 100;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..5341.85 rows=94 width=39) (actual time=0.451..16.845 rows=99 loops=1)
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Seq Scan on events  (cost=0.00..4332.45 rows=55 width=39) (actual time=0.003..5.204 rows=50 loops=2)
         Filter: (event_id < 100)
         Rows Removed by Filter: 137953
 Planning Time: 0.083 ms
 Execution Time: 16.861 ms
(8 rows)

Time: 17.215 ms
```

Query A:
In Query A, with index scans suppressed, the query planner is forced to use a parallel sequential scan to fetch rows where event_id < 100. This results in a significantly higher cost compared to when an index is available. Because the planner cannot use an index for direct access and must sequentially scan all 137, 953 rows, this adds significant overhead; the cost beginning at 1000.00 and increasing to 5341.85 is reflected in the large number of rows needing to be processed. The additional overhead explains the higher execution time (17.125ms) compared to the previous two execution times from (1) and (2) where the index scans were enabled (being 0.026ms and 0.025ms respectively).

Query B:
Query B similarly uses a sequential scan, as index scans are suppressed. Despite the large number of rows (275, 911) where event_id >= 100, the execution time (25.330ms) is relatively efficient since a sequential scan is more suitable for this sort of query that needs access to a substantial portion of the data.

Question 4 Q4 Transactions (4 Marks):

- In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with BEGIN and COMMIT commands.

(1) Begin Transaction --> Then Perform Query To Identify Item Which Is Currently Returned (1.5 Marks):

- I added an ORDER BY time_stamp DESC clause to ensure the most recent return is chosen.
- We do not commit in this first transaction

```
BEGIN;
SELECT item_id
FROM Events E
WHERE event_type = 'Return'
ORDER BY time_stamp DESC
LIMIT 1;
```

```
BEGIN
Time: 1.266 ms
a2=*# SELECT item_id
a2-*# FROM Events E
a2-*# WHERE event_type = 'Return'
a2-*# ORDER BY time_stamp DESC
a2-*# LIMIT 1;
    item_id
---------------
 UQ10000119361
(1 row)

Time: 43.314 ms
```

(2) Begin Another Transaction --> Attempt To Record Hold On That Item For Some Patron 14 Days In The Future, Then Commit (0.5 Marks):

```
BEGIN;
INSERT INTO EVENTS (event_id, patron_id, item_id, event_type, time_stamp, charge)
VALUES (NEXTVAL('events_event_id_seq'), 1, 'UQ10000119361', 'Hold', NOW() + INTERVAL '14
```

days', 0);
COMMIT;

```
a2=# BEGIN;
BEGIN
Time: 0.941 ms
a2=*# INSERT INTO EVENTS (event_id, patron_id, item_id, event_type, time_stamp, charge)
a2-*# VALUES (NEXTVAL('events_event_id_seq'), 1, 'UQ10000119361', 'Hold', NOW() + INTERVAL '14 days
', 0);
INSERT 0 1
Time: 8.393 ms
a2=*# COMMIT;
COMMIT
Time: 0.648 ms
```

(3) Attempt To Record Loan On Item For Different Patron (At Present Time), Then Commit (0.5 Marks):

BEGIN;
INSERT INTO EVENTS (event_id, patron_id, item_id, event_type, time_stamp, charge)
VALUES (NEXTVAL('events_event_id_seq'), 2, 'UQ10000119361', 'Loan', NOW(), 0);
COMMIT;

```
a2=*# BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
Time: 0.466 ms
a2=*# INSERT INTO EVENTS (event_id, patron_id, item_id, event_type, time_stamp, charge)
a2-*# VALUES (NEXTVAL('events_event_id_seq'), 2, 'UQ10000119361', 'Loan', NOW(), 0);
ERROR:  Item is currently held by another patron: item UQ10000119361
CONTEXT:  PL/pgSQL function udf_bi_loans() line 26 at RAISE
Time: 63.214 ms
a2=!# COMMIT;
ROLLBACK
Time: 0.240 ms
```

Explain What Happened Both In Database Terms And In Library Terms; Include A Schedule Or Precedence Graph Of The Transaction Events To Aid Explanation (1.5 Marks)
Hint: which event(s) were recorded for the item?

In Database Terms:

- Connection A began a transaction (T1) to identify the most recently returned item (item_id = UQ10000119361); while the query was executed successfully, Connection A does not commit this transaction T1 initially.
- Connection B then initiated a second transaction (T2) in order to place a 'Hold' on the same item (item_id = UQ10000119361) for patron_id = 1. This transaction was committed successfully, recording the 'Hold' event for the item.
- When Connection A (in T1) later attempted to commit a 'Loan' of the item to patron_id = 2, the operation failed because the item was already placed on 'Hold' by T2 in Connection B. PostgreSQL is serialisable and, therefore, prevents the 'Loan' while the item was still on 'Hold.
- Consequently, Transaction T1 was rolled back and no 'Loan' event was ever recorded.

In Library Terms:

- The library system successfully recorded the 'Hold' event for the item_id = UQ10000119361 for patron_id = 1; patron 1 will reserve the item for 14 days.
- The library system blocked the 'Loan' attempt for patron_id = 2 because the item was already held by another patron; this upholds the library's hold and loan policy.

Supporting Diagrams: