REPORT FILE INFS2200

ASSIGNMENT 2

NAME: NURUL BALQIS BINTI MOKHTAR

STUDENT ID: 48365363

Foreword

This report analyses the performance of SQL queries within a library database system, focusing on the impact of indexing, query execution plans, and transaction handling. The testing environment aimed to maintain consistent conditions, but variations in execution times may occur due to factors such as database caching, server load, and other system processes running concurrently.

Each section of the report begins by narrating the logic of the query execution, followed by a detailed analysis of access patterns, relational algebra concepts, and transaction behaviour. The results compare query plans with and without indexes, different isolation levels, and the use of views versus materialized views to highlight performance improvements or drawbacks.

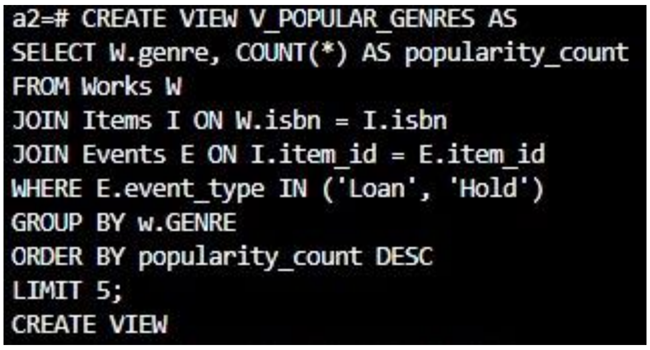
Overall, the report aims to demonstrate how database optimization techniques affect query efficiency, ensuring data consistency and fairness in concurrent operations within the system. The findings not only contribute to the theoretical understanding of SQL performance but also reflect practical applications in real-world database management.

Q1 Views

Q1.1 View A

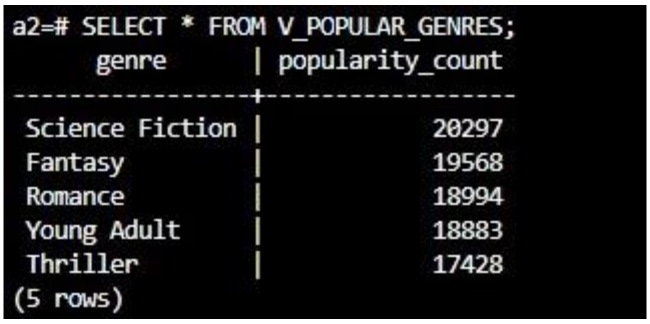
Query

```sql
CREATE VIEW V_POPULAR_GENRES AS
SELECT W.genre, COUNT(*) AS popularity_count
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 w.GENRE
ORDER BY popularity_count DESC
--determine 5
LIMIT 5;
--select view
SELECT * FROM V_POPULAR_GENRES;
--explain analyze view
EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;
```



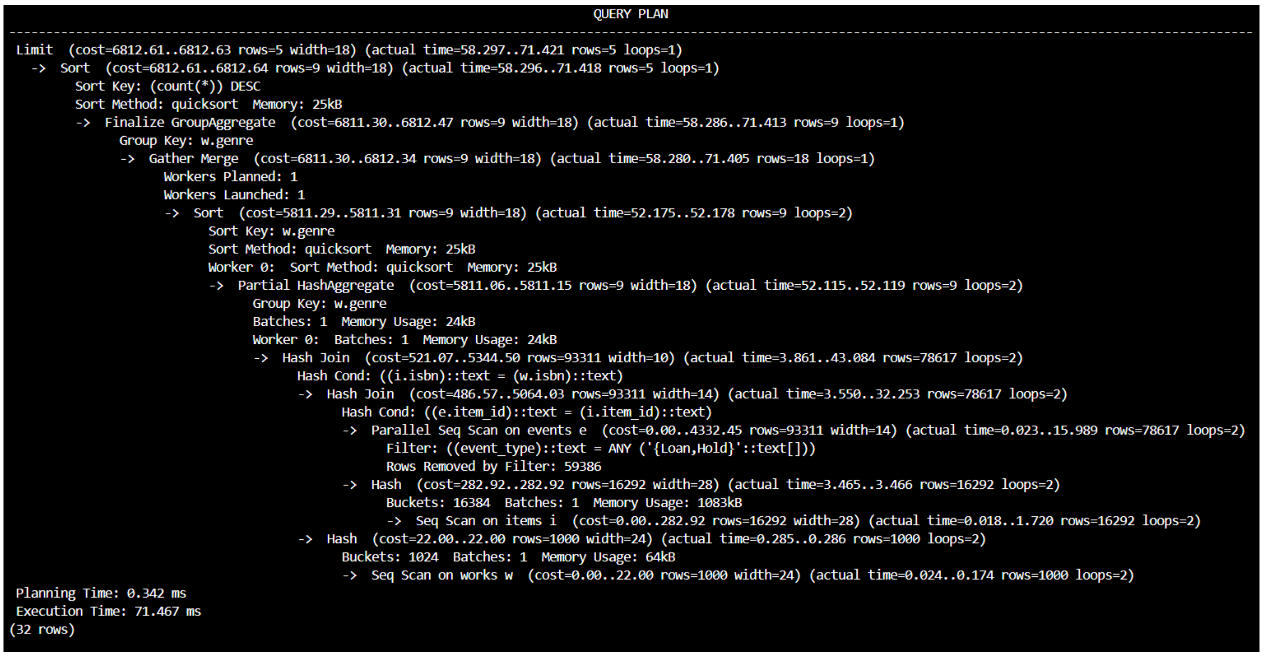
```
a2=# CREATE VIEW V_POPULAR_GENRES AS
SELECT W.genre, COUNT(*) AS popularity_count
FROM Works W
JOIN Items I ON W.isbn = I.isbn
JOIN Events E ON I.item_id = E.item_id
WHERE E.event_type IN ('Loan', 'Hold')
GROUP BY w.GENRE
ORDER BY popularity_count DESC
LIMIT 5;
CREATE VIEW
```

Figure A: CREATE VIEW V_POPULAR_GENRES



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

Figure B: SELECT VIEW V_POPULAR_GENRES



```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=6812.61..6812.63 rows=5 width=18) (actual time=58.297..71.421 rows=5 loops=1)
   ->  Sort  (cost=6812.61..6812.64 rows=9 width=18) (actual time=58.296..71.418 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=6811.30..6812.47 rows=9 width=18) (actual time=58.286..71.413 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=6811.30..6812.34 rows=9 width=18) (actual time=58.280..71.405 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=5811.29..5811.31 rows=9 width=18) (actual time=52.175..52.178 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=5811.06..5811.15 rows=9 width=18) (actual time=52.115..52.119 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..5344.50 rows=93311 width=10) (actual time=3.861..43.084 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.57..5064.03 rows=93311 width=14) (actual time=3.550..32.253 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=93311 width=14) (actual time=0.023..15.989 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=3.465..3.466 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=0.018..1.720 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.285..0.286 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=0.024..0.174 rows=1000 loops=2)
 Planning Time: 0.342 ms
 Execution Time: 71.467 ms
(32 rows)
```

Figure C: EXPLAIN ANALYZE SELECT VIEW V_POPULAR_GENRES

Explanation

First, the query filters for specific event_type values like 'Loan' or 'Hold' from the Events table, representing the selection operation in relational algebra. Next, it joins the results with the Works and Items tables on matching isbn and item_id, corresponding to the join operation. Finally, the query aggregates the results by counting genres, which is equivalent to the grouping operation in relational algebra.

In terms of relational algebra, this query follows the pattern of a selection (filtering rows based on event type and date), followed by a join operation (joining Events, Works, and Items on common fields), and finally a group by operation to aggregate the results by genre. By pushing the selection step early, the planner can reduce the number of rows involved in the join, improving query efficiency.

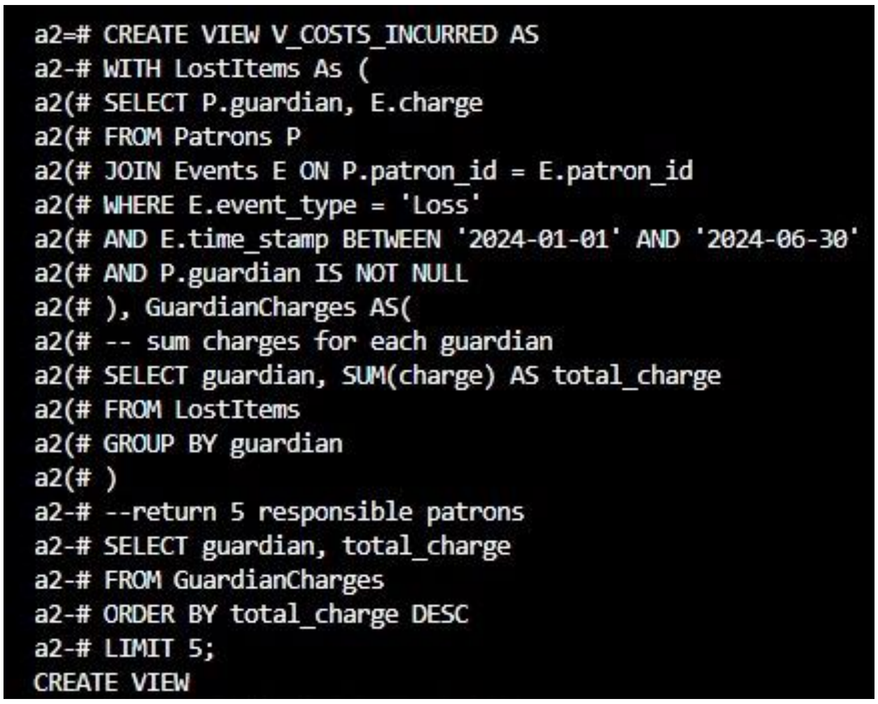
The query took 71.467 ms to run. That's pretty quick for a database query, but there's room to optimize it further. It took 0.342 ms for PostgreSQL to figure out how to run the query (planning time), meaning almost all of the time is spent in executing the query itself. PostgreSQL gives a cost estimate of 6812.63 for this query. This number is PostgreSQL's internal way of guessing how costly the query is in terms of database resources (CPU, memory, disk). While this is useful for comparison between different plans, the actual time (71 ms) is much lower than the estimated cost suggests. The query returns 5 rows, thanks to the LIMIT 5 clause. However, the database had to work through a lot more data — it scanned through 93311 rows from the Events table and 16292 rows from the Works table to get those results. The query filters for events where the event_type is either 'Loan' or 'Hold', meaning it’s only interested in items that were borrowed or held. This reduces the dataset but not enough to avoid scanning lots of rows.

Q1.2 View B

<u>Query</u>

```
--create view
CREATE VIEW V_COSTS_INCURRED AS
WITH LostItems As (
SELECT P.guardian, E.charge
FROM Patrons P
JOIN Events E ON P.patron_id = E.patron_id
WHERE E.event_type = 'Loss'
AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
AND P.guardian IS NOT NULL
), GuardianCharges AS(
-- sum charges for each guardian
SELECT guardian, SUM(charge) AS total_charge
FROM LostItems
GROUP BY guardian
)
--return 5 responsible patrons
SELECT guardian, total_charge
```

```sql
FROM GuardianCharges
ORDER BY total_charge DESC
LIMIT 5;
```



```
a2=# CREATE VIEW V_COSTS_INCURRED AS
a2-# WITH LostItems As (
a2(# SELECT P.guardian, E.charge
a2(# FROM Patrons P
a2(# JOIN Events E ON P.patron_id = E.patron_id
a2(# WHERE E.event_type = 'Loss'
a2(# AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
a2(# AND P.guardian IS NOT NULL
a2(# ), GuardianCharges AS(
a2(# -- sum charges for each guardian
a2(# SELECT guardian, SUM(charge) AS total_charge
a2(# FROM LostItems
a2(# GROUP BY guardian
a2(# )
a2-# --return 5 responsible patrons
a2-# SELECT guardian, total_charge
a2-# FROM GuardianCharges
a2-# ORDER BY total_charge DESC
a2-# LIMIT 5;
CREATE VIEW
```

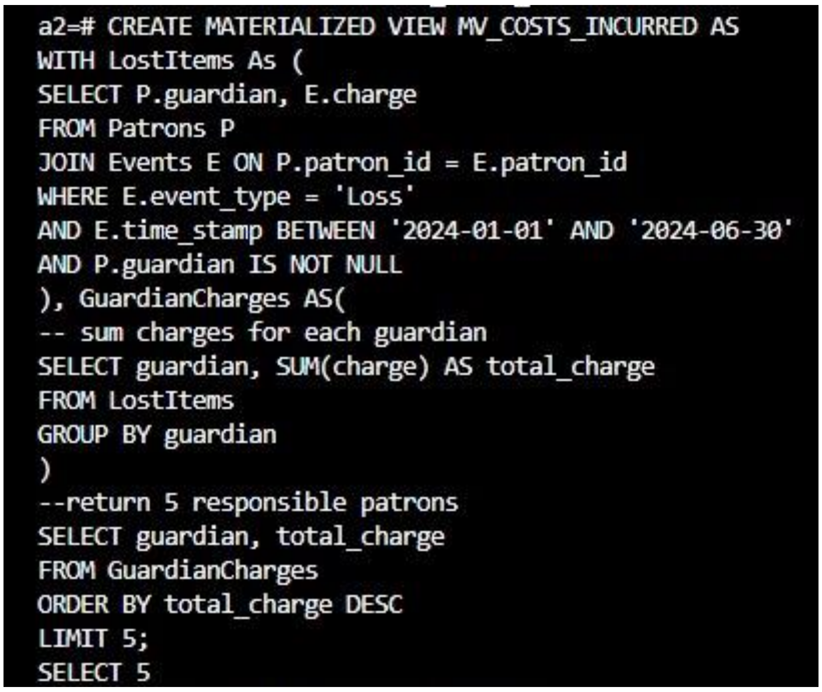
Figure D: CREATE VIEW V_COSTS_INCURRED

Q1.3 Materialised view

Query

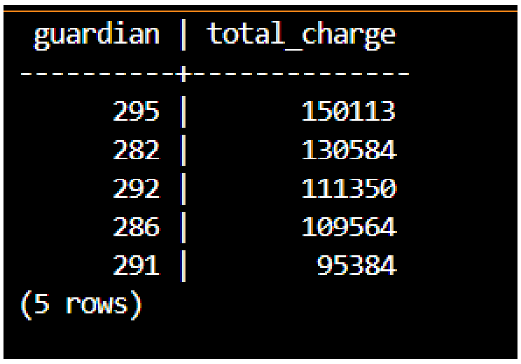
```sql
CREATE MATERIALIZED VIEW MV_COSTS_INCURRED AS
WITH LostItems As (
SELECT P.guardian, E.charge
FROM Patrons P
JOIN Events E ON P.patron_id = E.patron_id
WHERE E.event_type = 'Loss'
AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
AND P.guardian IS NOT NULL
), GuardianCharges AS(
-- sum charges for each guardian
SELECT guardian, SUM(charge) AS total_charge
FROM LostItems
GROUP BY guardian
)
--return 5 responsible patrons
SELECT guardian, total_charge
FROM GuardianCharges
ORDER BY total_charge DESC
LIMIT 5;
```

```
--select view
SELECT * FROM V_COSTS_INCURRED;
EXPLAIN ANALYZE SELECT * FROM V_COSTS_INCURRED;
--select materialized view
SELECT * FROM MV_COSTS_INCURRED;
EXPLAIN ANALYZE SELECT * FROM MV_COSTS_INCURRED;
```



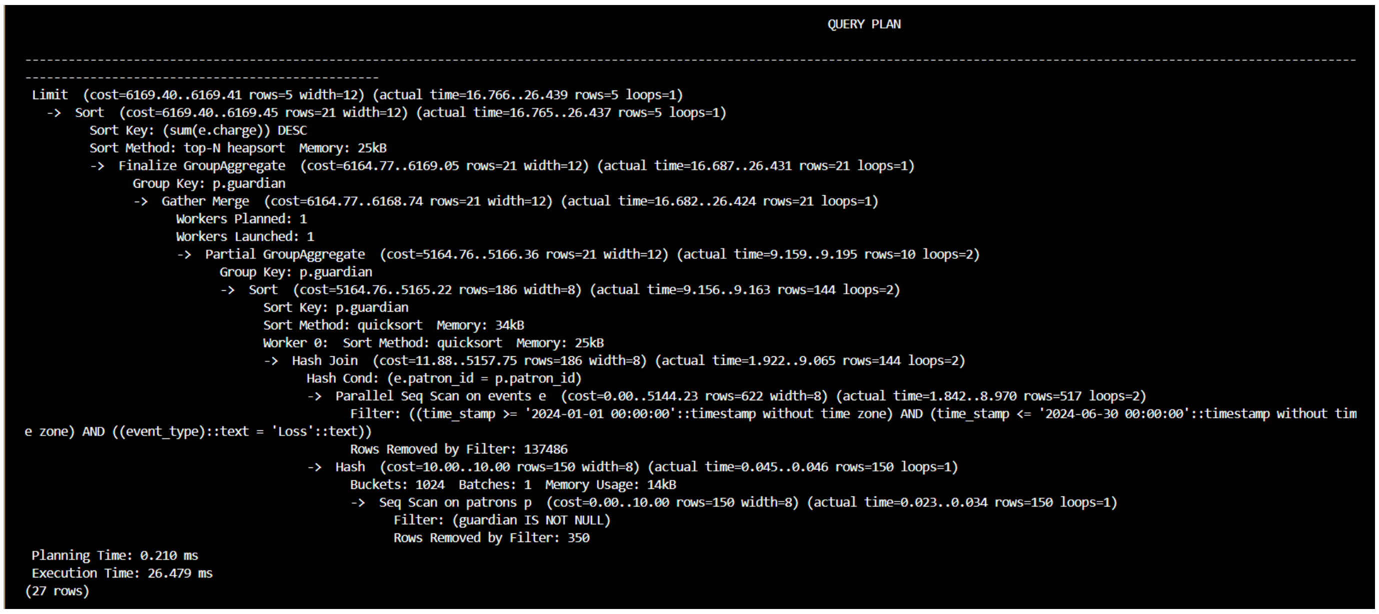
```
a2=# CREATE MATERIALIZED VIEW MV_COSTS_INCURRED AS
WITH LostItems As (
SELECT P.guardian, E.charge
FROM Patrons P
JOIN Events E ON P.patron_id = E.patron_id
WHERE E.event_type = 'Loss'
AND E.time_stamp BETWEEN '2024-01-01' AND '2024-06-30'
AND P.guardian IS NOT NULL
), GuardianCharges AS(
-- sum charges for each guardian
SELECT guardian, SUM(charge) AS total_charge
FROM LostItems
GROUP BY guardian
)
--return 5 responsible patrons
SELECT guardian, total_charge
FROM GuardianCharges
ORDER BY total_charge DESC
LIMIT 5;
SELECT 5
```

Figure E: SELECT 5



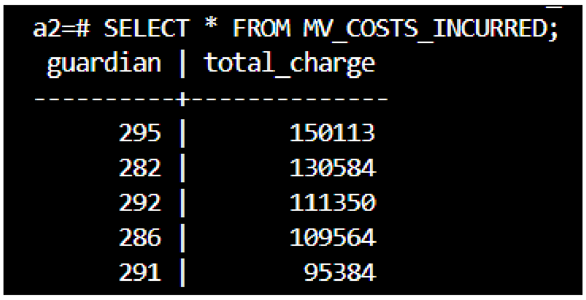
```
 guardian | total_charge
----------+--------------
      295 |       150113
      282 |       130584
      292 |       111350
      286 |       109564
      291 |        95384
(5 rows)
```

Figure F: SELECT FROM VIEW V_COSTS_INCURRED



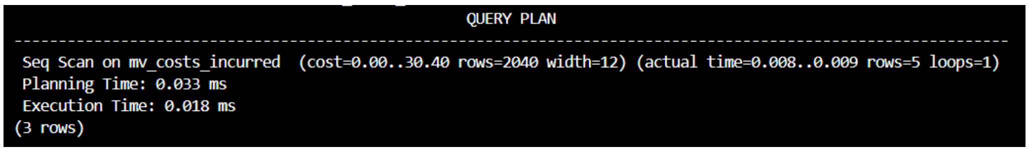
```
                                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------
 Limit  (cost=6169.40..6169.41 rows=5 width=12) (actual time=16.766..26.439 rows=5 loops=1)
   ->  Sort  (cost=6169.40..6169.45 rows=21 width=12) (actual time=16.765..26.437 rows=5 loops=1)
         Sort Key: (sum(e.charge)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=6164.77..6169.05 rows=21 width=12) (actual time=16.687..26.431 rows=21 loops=1)
               Group Key: p.guardian
               ->  Gather Merge  (cost=6164.77..6168.74 rows=21 width=12) (actual time=16.682..26.424 rows=21 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Partial GroupAggregate  (cost=5164.76..5166.36 rows=21 width=12) (actual time=9.159..9.195 rows=10 loops=2)
                           Group Key: p.guardian
                           ->  Sort  (cost=5164.76..5165.22 rows=186 width=8) (actual time=9.156..9.163 rows=144 loops=2)
                                 Sort Key: p.guardian
                                 Sort Method: quicksort  Memory: 34kB
                                 Worker 0:  Sort Method: quicksort  Memory: 25kB
                                 ->  Hash Join  (cost=11.88..5157.75 rows=186 width=8) (actual time=1.922..9.065 rows=144 loops=2)
                                       Hash Cond: (e.patron_id = p.patron_id)
                                       ->  Parallel Seq Scan on events e  (cost=0.00..5144.23 rows=622 width=8) (actual time=1.842..8.970 rows=517 loops=2)
                                             Filter: ((time_stamp >= '2024-01-01 00:00:00'::timestamp without time zone) AND (time_stamp <= '2024-06-30 00:00:00'::timestamp without tim
e zone) AND ((event_type)::text = 'Loss'::text))
                                             Rows Removed by Filter: 137486
                                       ->  Hash  (cost=10.00..10.00 rows=150 width=8) (actual time=0.045..0.046 rows=150 loops=1)
                                             Buckets: 1024  Batches: 1  Memory Usage: 14kB
                                             ->  Seq Scan on patrons p  (cost=0.00..10.00 rows=150 width=8) (actual time=0.023..0.034 rows=150 loops=1)
                                                   Filter: (guardian IS NOT NULL)
                                                   Rows Removed by Filter: 350
 Planning Time: 0.210 ms
 Execution Time: 26.479 ms
(27 rows)
```

Figure G: EXPLAIN ANALYZE SELECT FROM VIEW V_COSTS_INCURRED



```
a2=# SELECT * FROM MV_COSTS_INCURRED;
 guardian | total_charge
----------+--------------
      295 |       150113
      282 |       130584
      292 |       111350
      286 |       109564
      291 |        95384
```

Figure H: SELECT FROM MATERIALIZED VIEW



```
                                               QUERY PLAN
------------------------------------------------------------------------------------------------------
 Seq Scan on mv_costs_incurred  (cost=0.00..30.40 rows=2040 width=12) (actual time=0.008..0.009 rows=5 loops=1)
 Planning Time: 0.033 ms
 Execution Time: 0.018 ms
(3 rows)
```

Figure I: EXPLAIN ANALYZE SELECT FROM MATERIALIZED VIEW

Explanation

*SELECT * FROM V_COSTS_INCURRED:*

The execution time is typically slower because the query fetches fresh data each time it is run. It processes data dynamically, recomputing the result every time the view is accessed. Next, the cost is higher, as the query plan has to join and aggregate multiple tables (Patrons, Events, etc.) each time. Costs would include the CPU, memory, and I/O operations to re-evaluate the view. Then, 5 rows are fetched, as indicated by the LIMIT 5 clause in the query. The performance here is lower because it needs to calculate the result from the base tables on each execution.

*SELECT * FROM MV_COSTS_INCURRED:*

The execution time is super-fast which is 0.018 ms. This is because a materialized view stores the result of the query physically. Instead of recalculating the results each time, it fetches the precomputed data. Then, it costs 0.00..30.40, which is much lower than the actual view. The lower cost comes from the fact that the query only retrieves the precomputed data from the materialized view without needing to recompute joins or aggregations. It also fetches same 5 rows, but faster due to the stored result.

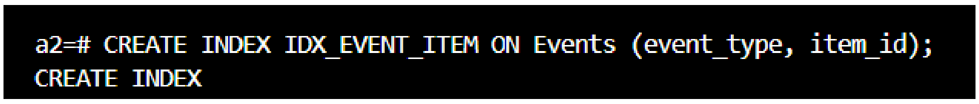
To conclude, materialized view is much more efficient due to precomputed data, with a lower cost and faster execution, meanwhile a view recalculates the data every time, leading to higher cost and slower performance, though it ensures real-time accuracy.

Q2 Indexes and performance

Q2.1 Basic index

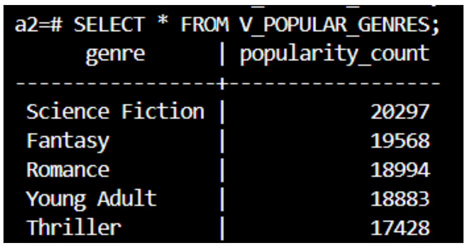
Query

```
--create index

CREATE INDEX IDX_EVENT_ITEM ON Events (event_type, item_id);
--select view
SELECT * FROM V_POPULAR_GENRES;
--explain anylyze select view
EXPLAIN ANALYZE SELECT * FROM V_POPULAR_GENRES;
```



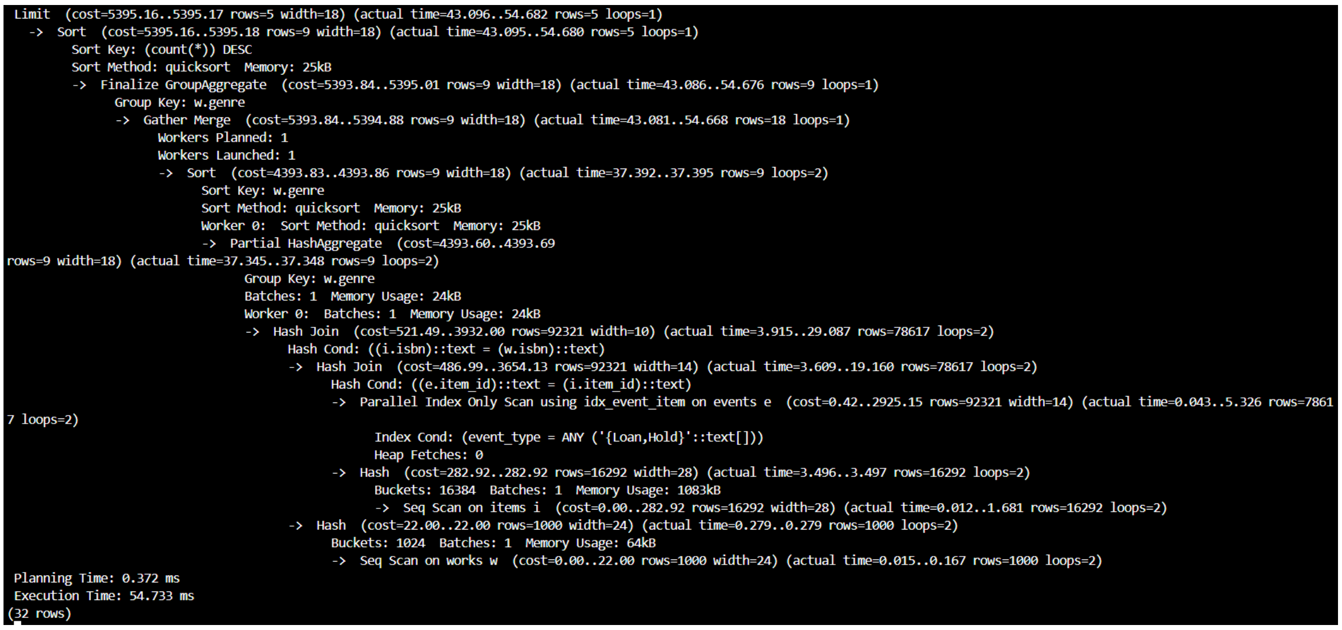
```
a2=# CREATE INDEX IDX_EVENT_ITEM ON Events (event_type, item_id);
CREATE INDEX
```

Figure J: CREATE INDEX IDX_EVENT_ITEM



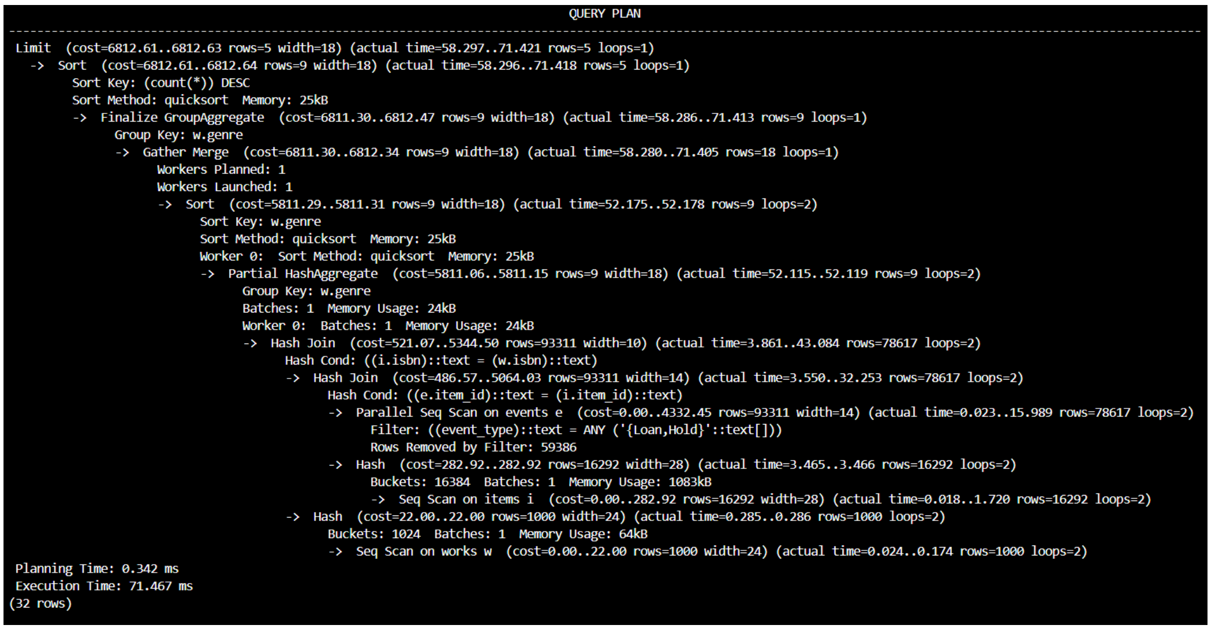
```
a2=# SELECT * FROM V_POPULAR_GENRES;
      genre      | popularity_count
-----------------+------------------
 Science Fiction |            20297
 Fantasy         |            19568
 Romance         |            18994
 Young Adult     |            18883
 Thriller        |            17428
```

Figure K: SELECT VIEW V_POPULAR_GENRES



```
 Limit  (cost=5395.16..5395.17 rows=5 width=18) (actual time=43.096..54.682 rows=5 loops=1)
   ->  Sort  (cost=5395.16..5395.18 rows=9 width=18) (actual time=43.095..54.680 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=5393.84..5395.01 rows=9 width=18) (actual time=43.086..54.676 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=5393.84..5394.88 rows=9 width=18) (actual time=43.081..54.668 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=4393.83..4393.86 rows=9 width=18) (actual time=37.392..37.395 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=4393.60..4393.69
rows=9 width=18) (actual time=37.345..37.348 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..3932.00 rows=92321 width=10) (actual time=3.915..29.087 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.99..3654.13 rows=92321 width=14) (actual time=3.609..19.160 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..2925.15 rows=92321 width=14) (actual time=0.043..5.326 rows=7861
7 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=3.496..3.497 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=0.012..1.681 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.279..0.279 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=0.015..0.167 rows=1000 loops=2)
 Planning Time: 0.372 ms
 Execution Time: 54.733 ms
(32 rows)
```

Figure L: QUERY PLAN WITH INDEX



```
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=6812.61..6812.63 rows=5 width=18) (actual time=58.297..71.421 rows=5 loops=1)
   ->  Sort  (cost=6812.61..6812.64 rows=9 width=18) (actual time=58.296..71.418 rows=5 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: quicksort  Memory: 25kB
         ->  Finalize GroupAggregate  (cost=6811.30..6812.47 rows=9 width=18) (actual time=58.286..71.413 rows=9 loops=1)
               Group Key: w.genre
               ->  Gather Merge  (cost=6811.30..6812.34 rows=9 width=18) (actual time=58.280..71.405 rows=18 loops=1)
                     Workers Planned: 1
                     Workers Launched: 1
                     ->  Sort  (cost=5811.29..5811.31 rows=9 width=18) (actual time=52.175..52.178 rows=9 loops=2)
                           Sort Key: w.genre
                           Sort Method: quicksort  Memory: 25kB
                           Worker 0:  Sort Method: quicksort  Memory: 25kB
                           ->  Partial HashAggregate  (cost=5811.06..5811.15 rows=9 width=18) (actual time=52.115..52.119 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..5344.50 rows=93311 width=10) (actual time=3.861..43.084 rows=78617 loops=2)
                                       Hash Cond: ((i.isbn)::text = (w.isbn)::text)
                                       ->  Hash Join  (cost=486.57..5064.03 rows=93311 width=14) (actual time=3.550..32.253 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=93311 width=14) (actual time=0.023..15.989 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=3.465..3.466 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=0.018..1.720 rows=16292 loops=2)
                                       ->  Hash  (cost=22.00..22.00 rows=1000 width=24) (actual time=0.285..0.286 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=0.024..0.174 rows=1000 loops=2)
 Planning Time: 0.342 ms
 Execution Time: 71.467 ms
(32 rows)
```

Figure M: QUERY PLAN WITHOUT INDEX

Explanation

First, the query looks for relevant rows by applying filters on event_type and item_id. Without an index, PostgreSQL performs a sequential scan, reading each row in the table. With the index in place, it skips directly to the rows that match, using an index scan for faster access.

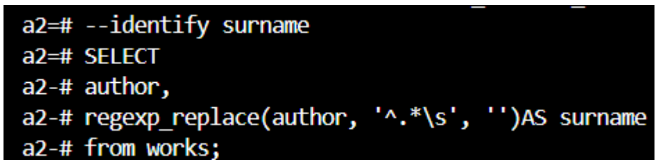
In terms of execution time, query with index, took **54.733 ms**. While without index query took **71.467 ms** to execute. The **indexed version is faster** by approximately 16.7 ms. This is because with the index in place, PostgreSQL can more efficiently locate the relevant rows instead of scanning through the entire table. Besides, the total cost without index is **6812.63**, while with index it's reduced to 6812.63. This drop in cost indicates that the database engine found a more efficient way to execute the query after the index was added. Other than that, without index, the plan shows Sequential Scans on the Events and Works tables. Sequential scans read every row in the table, which can be inefficient for large datasets. With index, the plan now shows a Parallel Index Only Scan on the Events table using the newly created idx_event_item. Index scans are faster because the database can directly locate relevant rows, skipping the full table scan.

In conclusion, as a result of the index, queries executed faster and costs were lower. The database uses the index to narrow down the search instead of scanning the whole table. Therefore, indexing event_type and item_id is essential for optimizing query performance. The access pattern changes significantly when the index is added. Instead of reading through every row (sequential scan), PostgreSQL can use an index scan, which narrows the search quickly by directly locating the relevant rows.

Q2.2 Function-based index

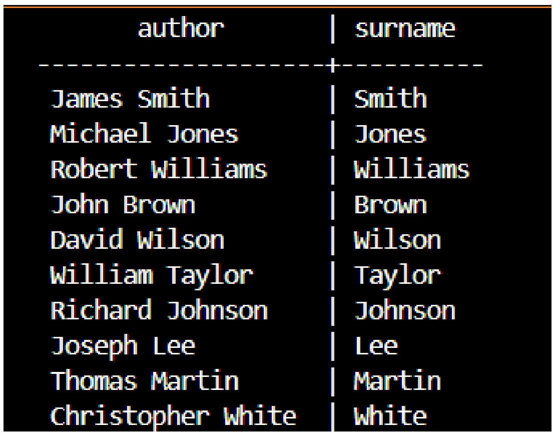
Query

```
--identify surname

SELECT
author,
regexp_replace(author, '^.*\s', '')AS surname
from works;
--explain analyze
EXPLAIN ANALYZE SELECT
author,
regexp_replace(author, '^.*\s', '')AS surname
from works;
--index to speed up queries
CREATE INDEX IDX_AUTHOR_SURNAME
ON WORKS ((REGEXP_REPLACE(AUTHOR, '^.*\s', '')));
--explain analyze
EXPLAIN ANALYZE SELECT
author,
regexp_replace(author, '^.*\s', '')AS surname
from works;
```



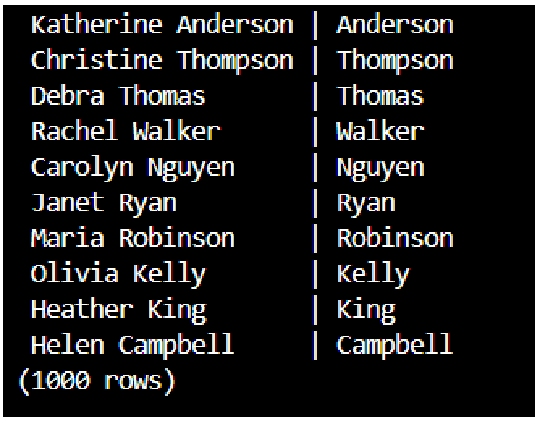
```
a2=# --identify surname
a2=# SELECT
a2-# author,
a2-# regexp_replace(author, '^.*\s', '')AS surname
a2-# from works;
```

Figure N: SELECT AUTHOR QUERY



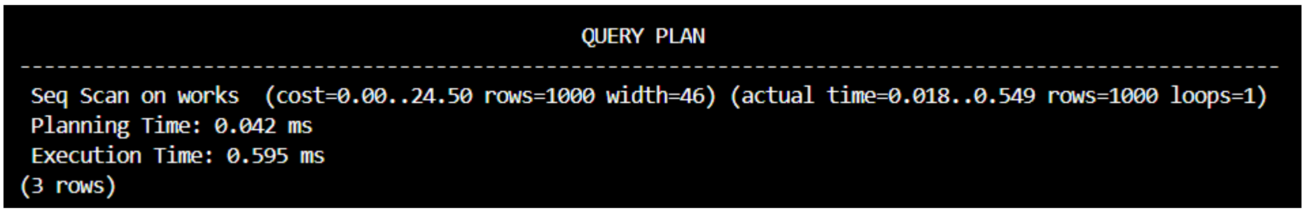
```
        author         | surname
-----------------------+----------
 James Smith           | Smith
 Michael Jones         | Jones
 Robert Williams       | Williams
 John Brown            | Brown
 David Wilson          | Wilson
 William Taylor        | Taylor
 Richard Johnson       | Johnson
 Joseph Lee            | Lee
 Thomas Martin         | Martin
 Christopher White     | White
```

Figure O: SELECT AUTHOR, first 10



```
 Katherine Anderson | Anderson
 Christine Thompson | Thompson
 Debra Thomas       | Thomas
 Rachel Walker      | Walker
 Carolyn Nguyen     | Nguyen
 Janet Ryan         | Ryan
 Maria Robinson     | Robinson
 Olivia Kelly       | Kelly
 Heather King       | King
 Helen Campbell     | Campbell
(1000 rows)
```

Figure P: SELECT AUTHOR, last 10

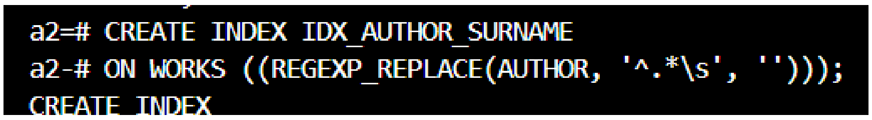


```
                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------
 Seq Scan on works  (cost=0.00..24.50 rows=1000 width=46) (actual time=0.018..0.549 rows=1000 loops=1)
 Planning Time: 0.042 ms
 Execution Time: 0.595 ms
(3 rows)
```

Figure Q: QUERY PLAN SELECT AUTHOR

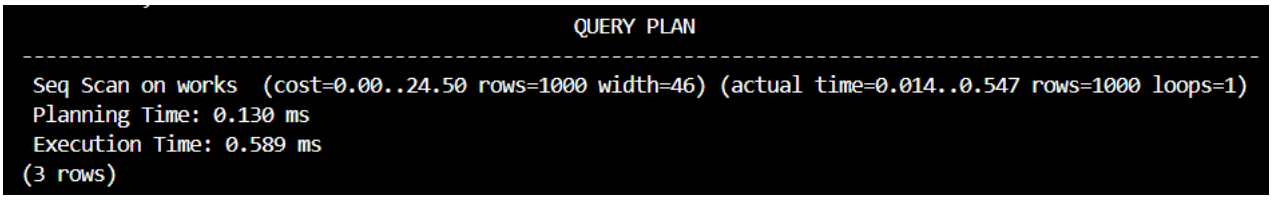
<u>Explanation</u>

The total time to execute the query was 0.595 ms. The planning time, or the time PostgreSQL took to decide how to execute the query, was 0.042 ms. This is relatively short because PostgreSQL opted for a simple sequential scan, which doesn't require complex planning. The cost of the query was estimated between **0.00 and 24.50**. Since the query involves scanning all rows in the table, the cost is proportional to the size of the Works table. 1000 rows were processed in this case. A sequential scan reads every row in the table, even though in larger datasets, only a small subset of rows might be relevant for specific queries.



```
a2=# CREATE INDEX IDX_AUTHOR_SURNAME
a2-# ON WORKS ((REGEXP_REPLACE(AUTHOR, '^.*\s', '')));
CREATE INDEX
```

Figure R: CREATE INDEX AUTHOR SURNAME



```
                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Seq Scan on works  (cost=0.00..24.50 rows=1000 width=46) (actual time=0.014..0.547 rows=1000 loops=1)
 Planning Time: 0.130 ms
 Execution Time: 0.589 ms
(3 rows)
```

Figure S: INDEXED QUERY PLAN SELECT AUTHOR

Explanation

After creating the index, the query plan was re-evaluated. The query plan still shows a sequential scan, indicating that the **index was not used**. The execution time is 0.589 ms which is only slightly faster than before, while the planning time is higher now, which is 0.130 ms. The cost and the rows fetched is exactly the same as before, which are 0.00 to 24.50 and 1000 rows respectively.

The execution time improved slightly, but the query cost and scan type did not change, meaning the index was not utilized in this specific query. It maybe because cost shown in the query plan (0.00..24.50) is relatively low. This low cost likely reflects that PostgreSQL thinks it's more efficient to just scan the table rather than use the index.

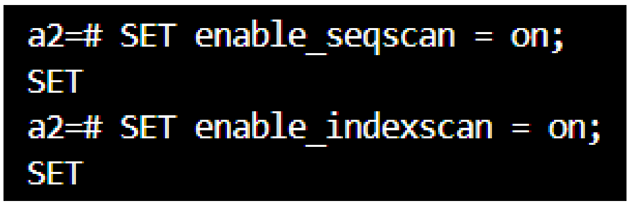
Q3 Indexes and query planning

Query

```
--(1) both enabled

SET enable_seqscan = on;
SET enable_indexscan = on;
--select
SELECT * FROM EVENTS WHERE EVENT_ID < 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID < 100;
SELECT * FROM EVENTS WHERE EVENT_ID >= 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID >= 100;

--reset
RESET enable_seqscan;
RESET enable_indexscan;

--(2) index scans enabled and sequential scans suppressed
SET enable_seqscan = off;
SET enable_indexscan = on;
```

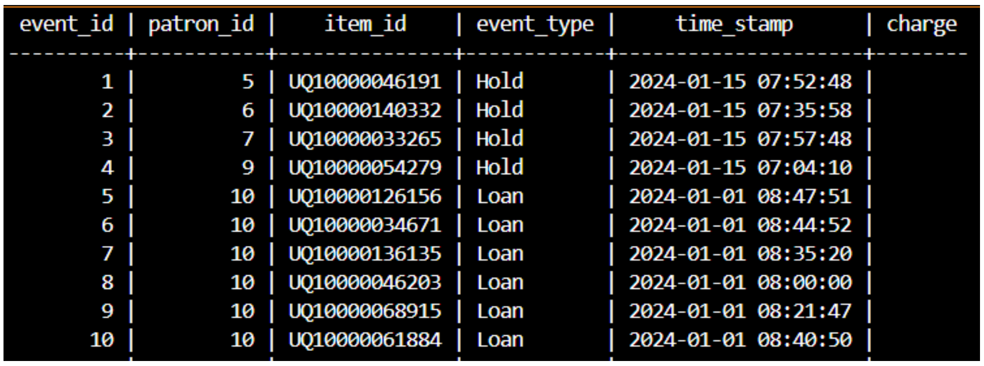
```
--select
SELECT * FROM EVENTS WHERE EVENT_ID < 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID < 100;
SELECT * FROM EVENTS WHERE EVENT_ID >= 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID >= 100;

--reset
RESET enable_seqscan;
RESET enable_indexscan;

--(3) index scans suppressed and sequential scans enabled
SET enable_seqscan = on;
SET enable_indexscan = off;
--select
SELECT * FROM EVENTS WHERE EVENT_ID < 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID < 100;
SELECT * FROM EVENTS WHERE EVENT_ID >= 100;
--explain analyze
EXPLAIN ANALYZE SELECT * FROM EVENTS WHERE EVENT_ID >= 100;
--reset
RESET enable_seqscan;
RESET enable_indexscan;
```

ENABLE INDEX AND SEQUENTIAL SCANS



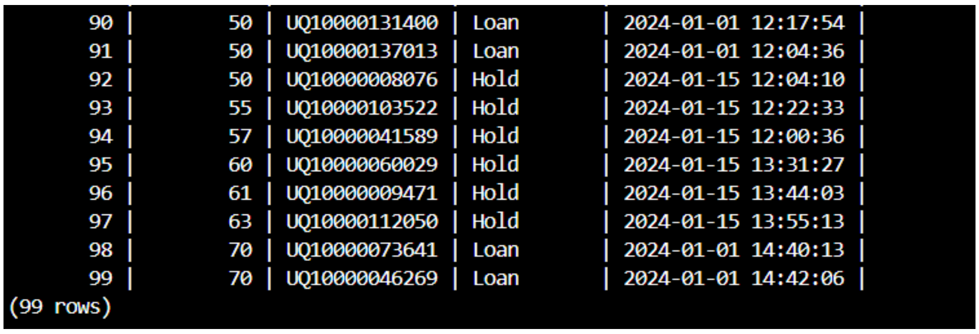
```
a2=# SET enable_seqscan = on;
SET
a2=# SET enable_indexscan = on;
SET
```

Figure T: ENABLE INDEX AND SEQUENTIAL SCANS



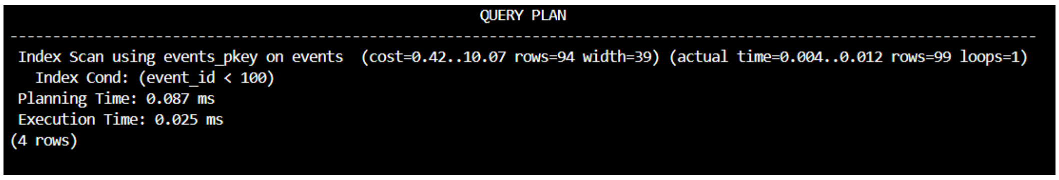
```
 event_id | patron_id |    item_id    | event_type |     time_stamp      | charge
----------+-----------+---------------+------------+---------------------+--------
        1 |         5 | UQ10000046191 | Hold       | 2024-01-15 07:52:48 |
        2 |         6 | UQ10000140332 | Hold       | 2024-01-15 07:35:58 |
        3 |         7 | UQ10000033265 | Hold       | 2024-01-15 07:57:48 |
        4 |         9 | UQ10000054279 | Hold       | 2024-01-15 07:04:10 |
        5 |        10 | UQ10000126156 | Loan       | 2024-01-01 08:47:51 |
        6 |        10 | UQ10000034671 | Loan       | 2024-01-01 08:44:52 |
        7 |        10 | UQ10000136135 | Loan       | 2024-01-01 08:35:20 |
        8 |        10 | UQ10000046203 | Loan       | 2024-01-01 08:00:00 |
        9 |        10 | UQ10000068915 | Loan       | 2024-01-01 08:21:47 |
       10 |        10 | UQ10000061884 | Loan       | 2024-01-01 08:40:50 |
```

Figure U: Both enabled, < 100, first 10



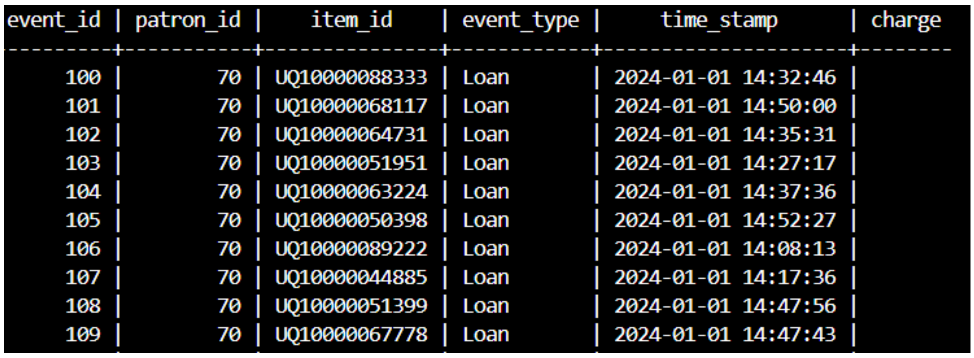
```
      90 |        50 | UQ10000131400 | Loan       | 2024-01-01 12:17:54 |
      91 |        50 | UQ10000137013 | Loan       | 2024-01-01 12:04:36 |
      92 |        50 | UQ10000008076 | Hold       | 2024-01-15 12:04:10 |
      93 |        55 | UQ10000103522 | Hold       | 2024-01-15 12:22:33 |
      94 |        57 | UQ10000041589 | Hold       | 2024-01-15 12:00:36 |
      95 |        60 | UQ10000060029 | Hold       | 2024-01-15 13:31:27 |
      96 |        61 | UQ10000009471 | Hold       | 2024-01-15 13:44:03 |
      97 |        63 | UQ10000112050 | Hold       | 2024-01-15 13:55:13 |
      98 |        70 | UQ10000073641 | Loan       | 2024-01-01 14:40:13 |
      99 |        70 | UQ10000046269 | Loan       | 2024-01-01 14:42:06 |
(99 rows)
```

Figure V: Both enabled, < 100, last 10



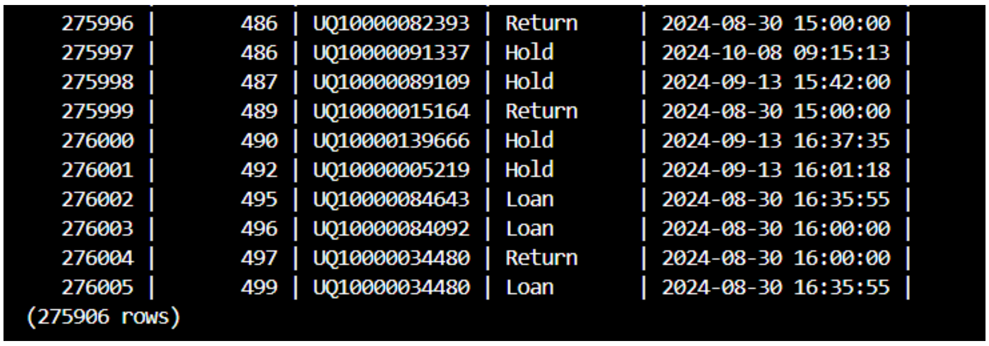
```
                                                         QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10.07 rows=94 width=39) (actual time=0.004..0.012 rows=99 loops=1)
   Index Cond: (event_id < 100)
 Planning Time: 0.087 ms
 Execution Time: 0.025 ms
(4 rows)
```

Figure W: QUERY PLAN Both enabled, < 100



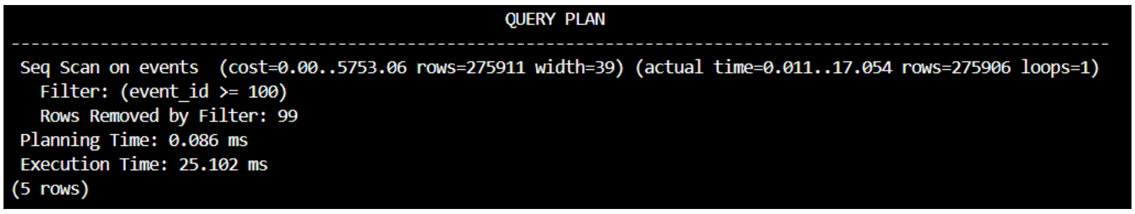
```
event_id | patron_id |    item_id    | event_type |      time_stamp      | charge
---------+-----------+---------------+------------+----------------------+--------
     100 |        70 | UQ10000088333 | Loan       | 2024-01-01 14:32:46 |
     101 |        70 | UQ10000068117 | Loan       | 2024-01-01 14:50:00 |
     102 |        70 | UQ10000064731 | Loan       | 2024-01-01 14:35:31 |
     103 |        70 | UQ10000051951 | Loan       | 2024-01-01 14:27:17 |
     104 |        70 | UQ10000063224 | Loan       | 2024-01-01 14:37:36 |
     105 |        70 | UQ10000050398 | Loan       | 2024-01-01 14:52:27 |
     106 |        70 | UQ10000089222 | Loan       | 2024-01-01 14:08:13 |
     107 |        70 | UQ10000044885 | Loan       | 2024-01-01 14:17:36 |
     108 |        70 | UQ10000051399 | Loan       | 2024-01-01 14:47:56 |
     109 |        70 | UQ10000067778 | Loan       | 2024-01-01 14:47:43 |
```

Figure X: Both enabled, >= 100, first 10



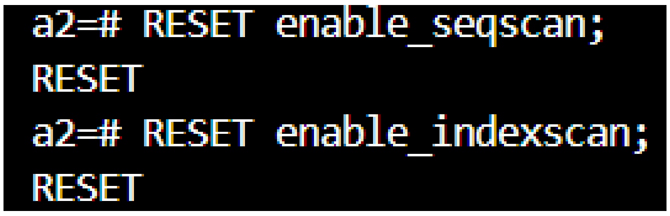
```
   275996 |       486 | UQ10000082393 | Return     | 2024-08-30 15:00:00 |
   275997 |       486 | UQ10000091337 | Hold       | 2024-10-08 09:15:13 |
   275998 |       487 | UQ10000089109 | Hold       | 2024-09-13 15:42:00 |
   275999 |       489 | UQ10000015164 | Return     | 2024-08-30 15:00:00 |
   276000 |       490 | UQ10000139666 | Hold       | 2024-09-13 16:37:35 |
   276001 |       492 | UQ10000005219 | Hold       | 2024-09-13 16:01:18 |
   276002 |       495 | UQ10000084643 | Loan       | 2024-08-30 16:35:55 |
   276003 |       496 | UQ10000084092 | Loan       | 2024-08-30 16:00:00 |
   276004 |       497 | UQ10000034480 | Return     | 2024-08-30 16:00:00 |
   276005 |       499 | UQ10000034480 | Loan       | 2024-08-30 16:35:55 |
(275906 rows)
```

Figure Y: Both enabled, >= 100, last 10



```
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Seq Scan on events  (cost=0.00..5753.06 rows=275911 width=39) (actual time=0.011..17.054 rows=275906 loops=1)
   Filter: (event_id >= 100)
   Rows Removed by Filter: 99
 Planning Time: 0.086 ms
 Execution Time: 25.102 ms
(5 rows)
```

Figure Z: QUERY PLAN Both enabled, >= 100



```
a2=# RESET enable_seqscan;
RESET
a2=# RESET enable_indexscan;
RESET
```

Figure AA: RESET scans settings

<u>Explanation</u>

In Query A, the planner first retrieves rows based on event_id, and then evaluates whether to use an index scan or sequential scan depending on the number of rows involved.

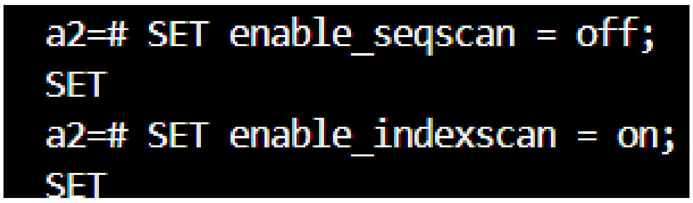
With both sequential and index scans enabled, the planner decides to use an index scan for smaller datasets (event_id < 100) and a sequential scan for larger datasets (event_id >= 100), reflecting an efficient access pattern. However, when only index scans are enabled, it is forced to use an index even for large datasets, which increases the cost.

In Query A, where event_id < 100, index scan was used. This is because since only 99 rows need to be fetched (a small subset of data), using an index is faster. The index lets PostgreSQL jump straight to the rows that meet the condition, avoiding a full table scan. The execution time is also very fast, as it took only 0.025 ms. It costs 0.42 to 10.07, reflecting the lower effort required to retrieve fewer rows. Here, PostgreSQL uses an index scan because the dataset is small (event_id < 100). The planner can efficiently retrieve relevant rows without scanning the entire table, showing an optimal access pattern for small data.

On the other hand, in query B, event_id >= 100, sequential scan was used because this query retrieves a much larger portion of the table (275,906 rows), so PostgreSQL prefers to scan the entire table. A sequential scan is more efficient for reading a large number of rows in one go rather than jumping back and forth using the index. The execution time is 25.102 ms which is longer because of the large number of rows. Lastly, it costs 0.00 to 5753.06, indicating the higher effort needed to process such a large result set. With a large number of rows to retrieve, the planner chooses a sequential scan. This access pattern allows for reading through the entire table in sequence, which is faster than using an index scan for large datasets.

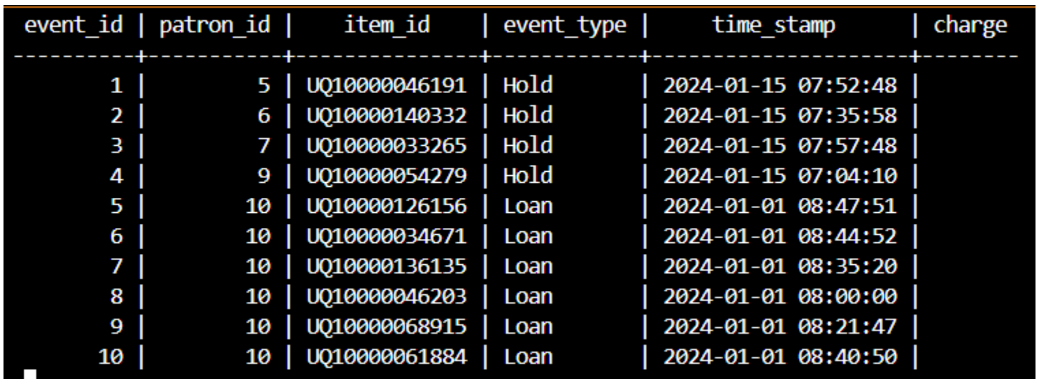
This plan is faster for small datasets because the planner uses an index scan for direct lookups. For large datasets, it switches to a sequential scan to avoid the overhead of accessing the index repeatedly, providing an optimal balance. Here, the planner efficiently switches between index scans for small datasets and sequential scans for large datasets. This avoids the overhead of index lookups for larger data, resulting in faster performance.

<u>INDEX SCANS ENABLED AND SEQUENTIAL SCANS SUPPRESSED</u>



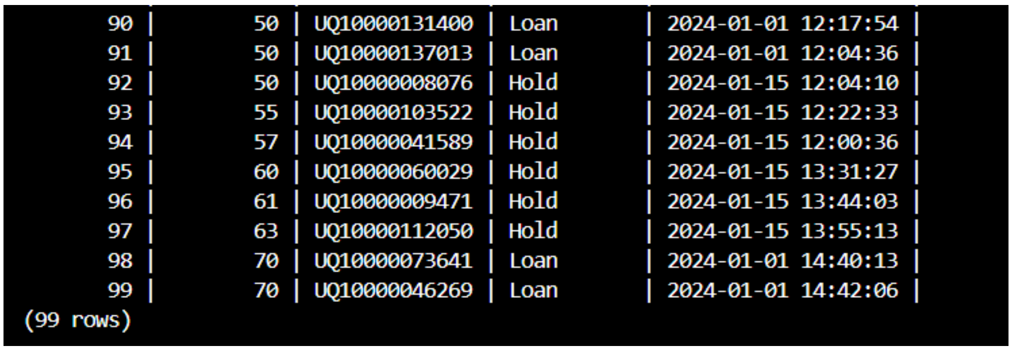
```
a2=# SET enable_seqscan = off;
SET
a2=# SET enable_indexscan = on;
SET
```

Figure BB: ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS



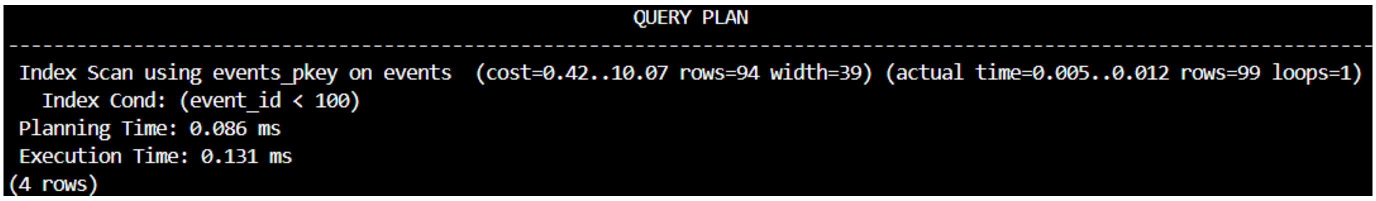
```
 event_id | patron_id |    item_id     | event_type |      time_stamp      | charge
----------+-----------+----------------+------------+---------------------+--------
        1 |         5 | UQ10000046191 | Hold       | 2024-01-15 07:52:48 |
        2 |         6 | UQ10000140332 | Hold       | 2024-01-15 07:35:58 |
        3 |         7 | UQ10000033265 | Hold       | 2024-01-15 07:57:48 |
        4 |         9 | UQ10000054279 | Hold       | 2024-01-15 07:04:10 |
        5 |        10 | UQ10000126156 | Loan       | 2024-01-01 08:47:51 |
        6 |        10 | UQ10000034671 | Loan       | 2024-01-01 08:44:52 |
        7 |        10 | UQ10000136135 | Loan       | 2024-01-01 08:35:20 |
        8 |        10 | UQ10000046203 | Loan       | 2024-01-01 08:00:00 |
        9 |        10 | UQ10000068915 | Loan       | 2024-01-01 08:21:47 |
       10 |        10 | UQ10000061884 | Loan       | 2024-01-01 08:40:50 |
```

Figure CC: SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, <100, first 10



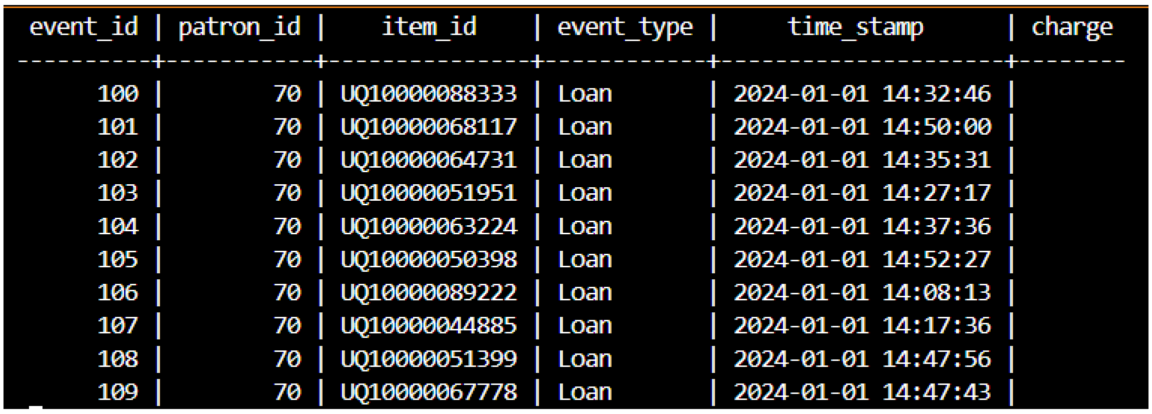
```
       90 |        50 | UQ10000131400 | Loan       | 2024-01-01 12:17:54 |
       91 |        50 | UQ10000137013 | Loan       | 2024-01-01 12:04:36 |
       92 |        50 | UQ10000008076 | Hold       | 2024-01-15 12:04:10 |
       93 |        55 | UQ10000103522 | Hold       | 2024-01-15 12:22:33 |
       94 |        57 | UQ10000041589 | Hold       | 2024-01-15 12:00:36 |
       95 |        60 | UQ10000060029 | Hold       | 2024-01-15 13:31:27 |
       96 |        61 | UQ10000009471 | Hold       | 2024-01-15 13:44:03 |
       97 |        63 | UQ10000112050 | Hold       | 2024-01-15 13:55:13 |
       98 |        70 | UQ10000073641 | Loan       | 2024-01-01 14:40:13 |
       99 |        70 | UQ10000046269 | Loan       | 2024-01-01 14:42:06 |
(99 rows)
```

Figure DD: SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, <100, last 10



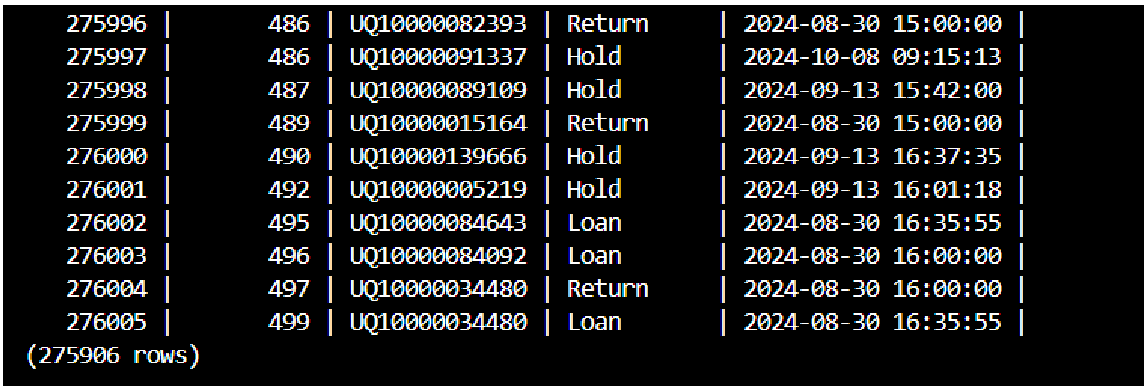
```
                                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10.07 rows=94 width=39) (actual time=0.005..0.012 rows=99 loops=1)
   Index Cond: (event_id < 100)
 Planning Time: 0.086 ms
 Execution Time: 0.131 ms
(4 rows)
```

Figure EE: Query plan SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, <100



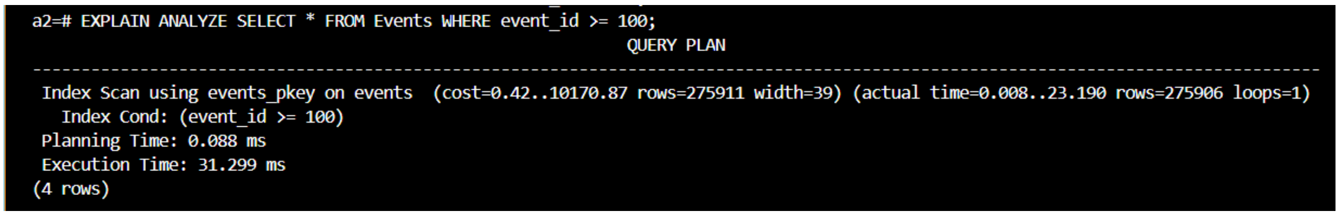
```
 event_id | patron_id |    item_id     | event_type |      time_stamp      | charge
----------+-----------+----------------+------------+---------------------+--------
      100 |        70 | UQ10000088333 | Loan       | 2024-01-01 14:32:46 |
      101 |        70 | UQ10000068117 | Loan       | 2024-01-01 14:50:00 |
      102 |        70 | UQ10000064731 | Loan       | 2024-01-01 14:35:31 |
      103 |        70 | UQ10000051951 | Loan       | 2024-01-01 14:27:17 |
      104 |        70 | UQ10000063224 | Loan       | 2024-01-01 14:37:36 |
      105 |        70 | UQ10000050398 | Loan       | 2024-01-01 14:52:27 |
      106 |        70 | UQ10000089222 | Loan       | 2024-01-01 14:08:13 |
      107 |        70 | UQ10000044885 | Loan       | 2024-01-01 14:17:36 |
      108 |        70 | UQ10000051399 | Loan       | 2024-01-01 14:47:56 |
      109 |        70 | UQ10000067778 | Loan       | 2024-01-01 14:47:43 |
```

Figure FF: SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, >= 100, first 10



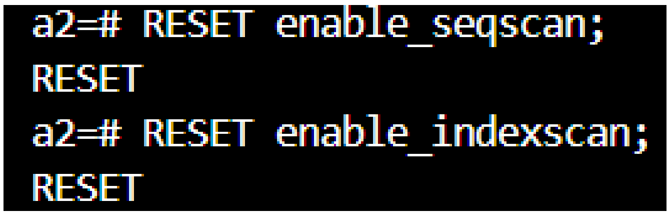
```
   275996 |       486 | UQ10000082393 | Return     | 2024-08-30 15:00:00 |
   275997 |       486 | UQ10000091337 | Hold       | 2024-10-08 09:15:13 |
   275998 |       487 | UQ10000089109 | Hold       | 2024-09-13 15:42:00 |
   275999 |       489 | UQ10000015164 | Return     | 2024-08-30 15:00:00 |
   276000 |       490 | UQ10000139666 | Hold       | 2024-09-13 16:37:35 |
   276001 |       492 | UQ10000005219 | Hold       | 2024-09-13 16:01:18 |
   276002 |       495 | UQ10000084643 | Loan       | 2024-08-30 16:35:55 |
   276003 |       496 | UQ10000084092 | Loan       | 2024-08-30 16:00:00 |
   276004 |       497 | UQ10000034480 | Return     | 2024-08-30 16:00:00 |
   276005 |       499 | UQ10000034480 | Loan       | 2024-08-30 16:35:55 |
(275906 rows)
```

Figure GG: SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, >= 100, last 10



```
a2=# EXPLAIN ANALYZE SELECT * FROM Events WHERE event_id >= 100;
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Index Scan using events_pkey on events  (cost=0.42..10170.87 rows=275911 width=39) (actual time=0.008..23.190 rows=275906 loops=1)
   Index Cond: (event_id >= 100)
 Planning Time: 0.088 ms
 Execution Time: 31.299 ms
(4 rows)
```

Figure HH: Query plan SELECT ENABLE INDEX SCAN, OFF SEQUENTIAL SCANS, >= 100



```
a2=# RESET enable_seqscan;
RESET
a2=# RESET enable_indexscan;
RESET
```

Figure II: RESET scans

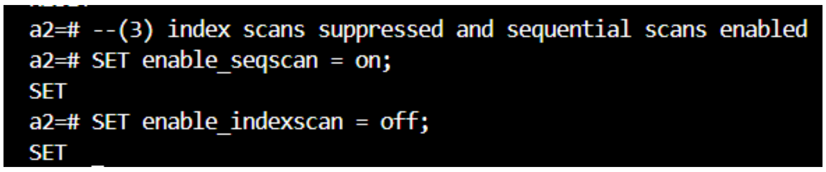
<u>Explanation</u>

In this case, for the Query A, event_id < 100, the execution time was 0.131 ms with a cost of 0.42 to 10.07. A total of 99 rows were fetched. PostgreSQL opted for an index scan because only a small number of rows were needed. This allowed it to directly access the required rows, avoiding a full table scan, making it faster for smaller data sets.

For the Query B, event_id >= 100, the Index Scan with Sequential Scan Suppressed took significantly longer with an execution time of 31.299 ms and a cost of 0.42 to 10107.87. In this case, 275,906 rows were fetched. Although many rows were involved, PostgreSQL was forced to use an index because sequential scans were disabled. Since this option was suppressed, the index scan was used, resulting in slower performance and higher costs due to the size of the data.

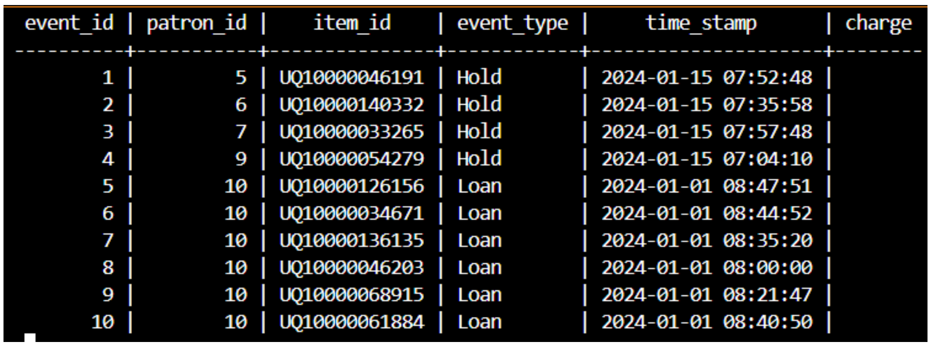
Here, the forced use of index scans slows down performance when dealing with large datasets, as it must repeatedly access the index for many rows. This is a less efficient approach for large datasets, which would benefit more from a sequential scan. Forcing index scans in large datasets increases execution time, as PostgreSQL has to look up every row in the index, even when it would be more efficient to perform a sequential scan.

INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED



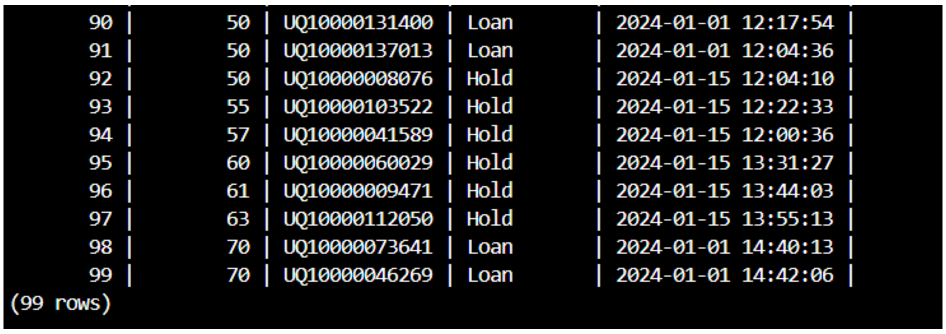
```
a2=# --(3) index scans suppressed and sequential scans enabled
a2=# SET enable_seqscan = on;
SET
a2=# SET enable_indexscan = off;
SET
```

Figure JJ: SET INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED



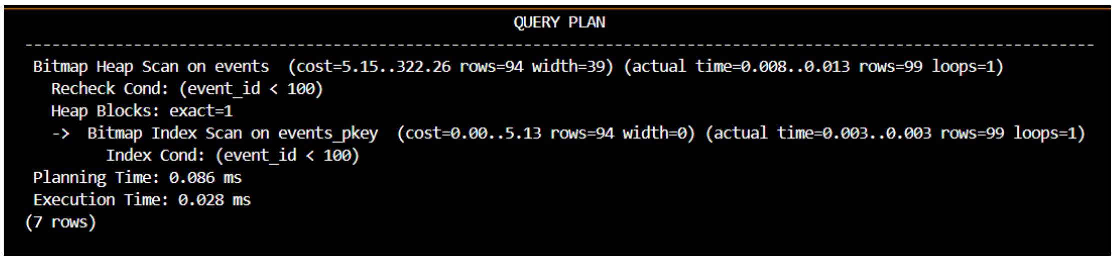
```
 event_id | patron_id |    item_id     | event_type |     time_stamp      | charge
----------+-----------+----------------+------------+---------------------+--------
        1 |         5 | UQ10000046191 | Hold       | 2024-01-15 07:52:48 |
        2 |         6 | UQ10000140332 | Hold       | 2024-01-15 07:35:58 |
        3 |         7 | UQ10000033265 | Hold       | 2024-01-15 07:57:48 |
        4 |         9 | UQ10000054279 | Hold       | 2024-01-15 07:04:10 |
        5 |        10 | UQ10000126156 | Loan       | 2024-01-01 08:47:51 |
        6 |        10 | UQ10000034671 | Loan       | 2024-01-01 08:44:52 |
        7 |        10 | UQ10000136135 | Loan       | 2024-01-01 08:35:20 |
        8 |        10 | UQ10000046203 | Loan       | 2024-01-01 08:00:00 |
        9 |        10 | UQ10000068915 | Loan       | 2024-01-01 08:21:47 |
       10 |        10 | UQ10000061884 | Loan       | 2024-01-01 08:40:50 |
```

Figure KK: SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, <100, first 10



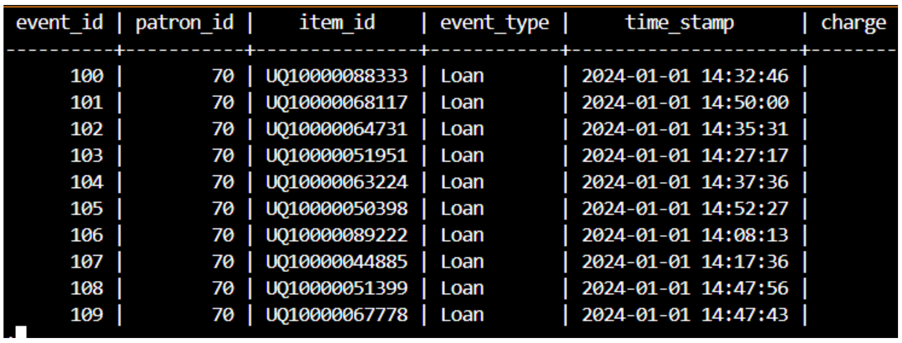
```
       90 |        50 | UQ10000131400 | Loan       | 2024-01-01 12:17:54 |
       91 |        50 | UQ10000137013 | Loan       | 2024-01-01 12:04:36 |
       92 |        50 | UQ10000008076 | Hold       | 2024-01-15 12:04:10 |
       93 |        55 | UQ10000103522 | Hold       | 2024-01-15 12:22:33 |
       94 |        57 | UQ10000041589 | Hold       | 2024-01-15 12:00:36 |
       95 |        60 | UQ10000060029 | Hold       | 2024-01-15 13:31:27 |
       96 |        61 | UQ10000009471 | Hold       | 2024-01-15 13:44:03 |
       97 |        63 | UQ10000112050 | Hold       | 2024-01-15 13:55:13 |
       98 |        70 | UQ10000073641 | Loan       | 2024-01-01 14:40:13 |
       99 |        70 | UQ10000046269 | Loan       | 2024-01-01 14:42:06 |
(99 rows)
```

Figure LL: SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, <100, last 10



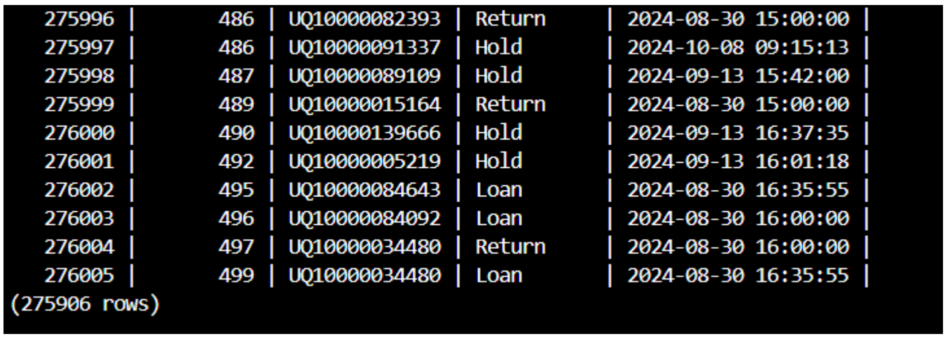
```
                                                       QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on events  (cost=5.15..322.26 rows=94 width=39) (actual time=0.008..0.013 rows=99 loops=1)
   Recheck Cond: (event_id < 100)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on events_pkey  (cost=0.00..5.13 rows=94 width=0) (actual time=0.003..0.003 rows=99 loops=1)
         Index Cond: (event_id < 100)
 Planning Time: 0.086 ms
 Execution Time: 0.028 ms
(7 rows)
```

Figure MM: Query plan SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, < 100



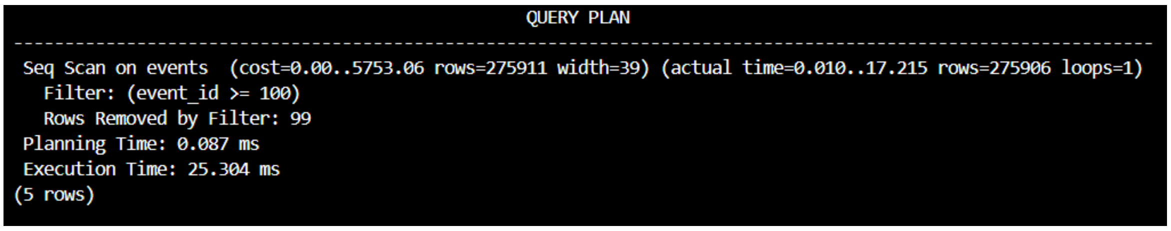
```
 event_id | patron_id |    item_id     | event_type |      time_stamp      | charge
----------+-----------+----------------+------------+----------------------+--------
      100 |        70 | UQ10000088333 | Loan       | 2024-01-01 14:32:46 |
      101 |        70 | UQ10000068117 | Loan       | 2024-01-01 14:50:00 |
      102 |        70 | UQ10000064731 | Loan       | 2024-01-01 14:35:31 |
      103 |        70 | UQ10000051951 | Loan       | 2024-01-01 14:27:17 |
      104 |        70 | UQ10000063224 | Loan       | 2024-01-01 14:37:36 |
      105 |        70 | UQ10000050398 | Loan       | 2024-01-01 14:52:27 |
      106 |        70 | UQ10000089222 | Loan       | 2024-01-01 14:08:13 |
      107 |        70 | UQ10000044885 | Loan       | 2024-01-01 14:17:36 |
      108 |        70 | UQ10000051399 | Loan       | 2024-01-01 14:47:56 |
      109 |        70 | UQ10000067778 | Loan       | 2024-01-01 14:47:43 |
```

Figure NN: SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, >= 100, first 10



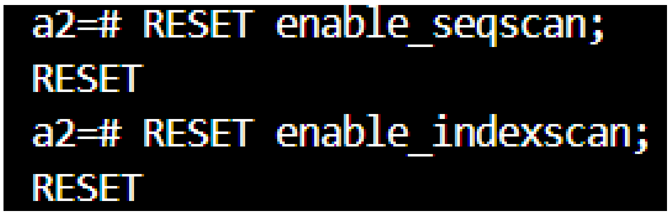
```
   275996 |       486 | UQ10000082393 | Return     | 2024-08-30 15:00:00 |
   275997 |       486 | UQ10000091337 | Hold       | 2024-10-08 09:15:13 |
   275998 |       487 | UQ10000089109 | Hold       | 2024-09-13 15:42:00 |
   275999 |       489 | UQ10000015164 | Return     | 2024-08-30 15:00:00 |
   276000 |       490 | UQ10000139666 | Hold       | 2024-09-13 16:37:35 |
   276001 |       492 | UQ10000005219 | Hold       | 2024-09-13 16:01:18 |
   276002 |       495 | UQ10000084643 | Loan       | 2024-08-30 16:35:55 |
   276003 |       496 | UQ10000084092 | Loan       | 2024-08-30 16:00:00 |
   276004 |       497 | UQ10000034480 | Return     | 2024-08-30 16:00:00 |
   276005 |       499 | UQ10000034480 | Loan       | 2024-08-30 16:35:55 |
(275906 rows)
```

Figure OO: SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, >= 100, last 10



```
                                                   QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on events  (cost=0.00..5753.06 rows=275911 width=39) (actual time=0.010..17.215 rows=275906 loops=1)
   Filter: (event_id >= 100)
   Rows Removed by Filter: 99
 Planning Time: 0.087 ms
 Execution Time: 25.304 ms
(5 rows)
```

Figure PP : Query plan SELECT INDEX SCANS SUPPRESSED AND SEQUENTIAL SCANS ENABLED, >= 100



```
a2=# RESET enable_seqscan;
RESET
a2=# RESET enable_indexscan;
RESET
```

Figure QQ: RESET scans

Explanation

Looking at Query A, when event_id < 100, the index scan is faster because it directly accesses the 99 rows that match the condition. The execution time is short (0.131 ms) because the planner avoids reading the entire table. Using the index is optimal for such small datasets because it minimizes the number of rows to scan.

However, in Query B, when event_id >= 100, the index scan becomes less efficient despite being forced. In this case, the index scan processes 275,906 rows, leading to a much longer execution time (31.299 ms) and higher costs. Sequential scans are typically better for larger datasets because they read the whole table efficiently. Here, forcing an index scan leads to slower performance due to the sheer size of the result set.

By forcing PostgreSQL to only use an index scan, even on large datasets, we can observe that this access pattern leads to higher costs and slower execution times, as it repeatedly looks up values in the index for many rows. In this plan, suppressing the index scan results in a sequential scan, which is optimal for large datasets but less efficient for small ones. The sequential scan reads every row in the table, which can be slower when only a small subset of data is needed.

Conclusion

Relational algebra here involves the selection of rows based on event_id, followed by an evaluation of the access pattern (either index scan or sequential scan) depending on the size of the dataset.

In conclusion, the planner's decision on whether to use an index scan or a sequential scan depends on the size of the dataset and the relative cost of each approach. For smaller datasets, index scans offer faster performance due to direct lookups, while sequential scans are better suited for larger datasets. Forcing one method over the other in suboptimal scenarios, like using an index scan for large datasets, can lead to slower performance and higher costs.

Q4 Transactions

Query

--first connection

```
BEGIN; --START TRANSACTION
--THIS PART WOULD TAKE A FEW MINUTES TO DISPLAY RESULT
SELECT
I.*
```

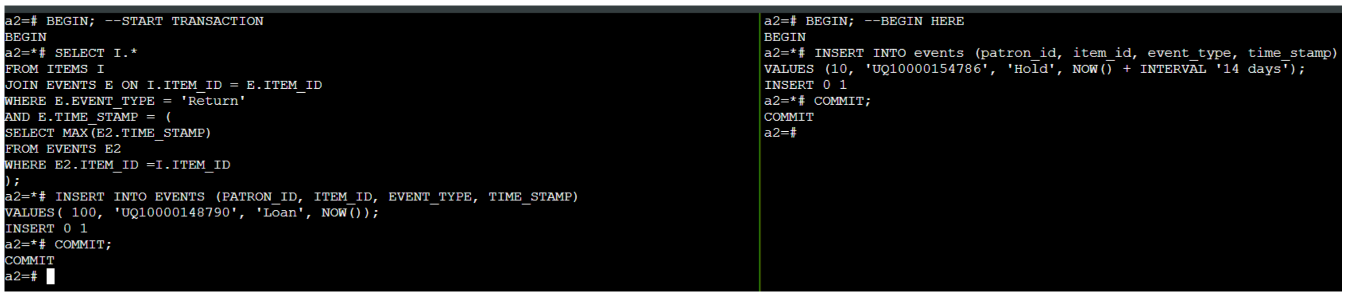
```sql
FROM ITEMS
I
JOIN EVENTS E ON I.ITEM_ID =
E.ITEM_ID
WHERE E.EVENT_TYPE =
'Return'
AND E.TIME_STAMP =
(
SELECT
MAX(E2.TIME_STAMP)

FROM EVENTS
E2
WHERE E2.ITEM_ID
=I.ITEM_ID
);
```

--second connection

```sql
BEGIN; --START
INSERT INTO events (patron_id, item_id, event_type,
time_stamp)
VALUES (10, 'UQ10000154786', 'Hold', NOW() + INTERVAL '14 days');
COMMIT;
```

--first connection

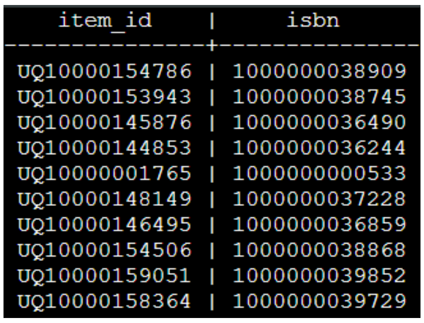
```sql
INSERT INTO EVENTS (PATRON_ID, ITEM_ID, EVENT_TYPE, TIME_STAMP)
VALUES( 100, 'UQ10000148790', 'Loan', NOW());
COMMIT;
```



```
a2=# BEGIN; --START TRANSACTION
BEGIN
a2=*# SELECT I.*
FROM ITEMS I
JOIN EVENTS E ON I.ITEM_ID = E.ITEM_ID
WHERE E.EVENT_TYPE = 'Return'
AND E.TIME_STAMP = (
SELECT MAX(E2.TIME_STAMP)
FROM EVENTS E2
WHERE E2.ITEM_ID =I.ITEM_ID
);
a2=*# INSERT INTO EVENTS (PATRON_ID, ITEM_ID, EVENT_TYPE, TIME_STAMP)
VALUES( 100, 'UQ10000148790', 'Loan', NOW());
INSERT 0 1
a2=*# COMMIT;
COMMIT
a2=#
```

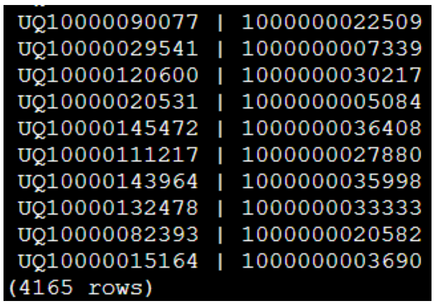
```
a2=# BEGIN; --BEGIN HERE
BEGIN
a2=*# INSERT INTO events (patron_id, item_id, event_type, time_stamp)
VALUES (10, 'UQ10000154786', 'Hold', NOW() + INTERVAL '14 days');
INSERT 0 1
a2=*# COMMIT;
COMMIT
a2=#
```

Figure RR: transactions



```
    item_id     |      isbn
----------------+----------------
 UQ10000154786 | 1000000038909
 UQ10000153943 | 1000000038745
 UQ10000145876 | 1000000036490
 UQ10000144853 | 1000000036244
 UQ10000001765 | 1000000000533
 UQ10000148149 | 1000000037228
 UQ10000146495 | 1000000036859
 UQ10000154506 | 1000000038868
 UQ10000159051 | 1000000039852
 UQ10000158364 | 1000000039729
```

Figure SS: SELECT I, first 10



```
 UQ10000090077 | 1000000022509
 UQ10000029541 | 1000000007339
 UQ10000120600 | 1000000030217
 UQ10000020531 | 1000000005084
 UQ10000145472 | 1000000036408
 UQ10000111217 | 1000000027880
 UQ10000143964 | 1000000035998
 UQ10000132478 | 1000000033333
 UQ10000082393 | 1000000020582
 UQ10000015164 | 1000000003690
(4165 rows)
```

Figure TT: SELECT I, last 10

<u>Schedule of transaction</u>

Time	First connection (T1)	Second connection (T2)	Description
t1	BEGIN		Start Transaction for first connection, T1
t2	SELECT query on returns		T1 searches for latest return events
t3		BEGIN	Start Transaction for second connection, T2
t4		INSERT ‘Hold’ event for item A	T2 places a hold on an item
t5		COMMIT	T2 commits, hold on item A is now visible
t6	INSERT ‘Loan’ event for item B		T1 attempts to loan item B
t7			If T1 tries loaning item A (on hold), it would be blocked
t8	COMMIT		T1 commits, loan of item B is successful

Explanation

First Connection:

The first connection starts a transaction to search for items that were recently returned. It uses a query that joins the items and events tables and looks for the most recent return event for each item. This process may take some time since the database must find the latest timestamp for every returned item. While this query is running, the database applies a read lock, which prevents other transactions from modifying the same data until the query finishes.

Second Connection:

While the first connection is still active, a second connection starts another transaction to place a hold on an item. The system inserts a new event in the events table, marking the item as on hold for 14 days. This ensures that no other user can borrow the item during this time. The database checks foreign key constraints to make sure the patron_id and item_id exist. Once committed, the hold becomes visible to other transactions.

Returning to the First Connection:

After the first connection completes its query, it tries to record a loan for another item. If the same item had been placed on hold by the second connection, the loan would be blocked. The database ensures that conflicting actions (like loaning a held item) are prevented through constraints or triggers.

Elaboration:

In database terms, these transactions demonstrate how transaction isolation levels manage concurrency. If both transactions used a high isolation level like SERIALIZABLE, the second connection would have to wait until the first connection's query finished to place a hold. This avoids issues like dirty reads or conflicting changes. With a lower isolation level, such as READ COMMITTED, the second connection can proceed without waiting, and any conflicts (like trying to loan a held item) are resolved through locks or constraints at commit time.

In a library setting, these actions represent typical operations. When a librarian places a hold on an item, it ensures no other user can borrow it during that period. If another user tries to loan the item while it’s on hold, the system blocks the loan to avoid conflicts. This coordination ensures fairness, preventing simultaneous actions that could result in double booking or user frustration.