GUC
German University in Cairo
الجامعة الألمانية بالقاهرة

CSEN604

Project 2

Prepared by:

Nada Alaa 52-4138

Mohga El Barbary 55-1706

Sara Wael 55-0768

Yara Ahmed 55-12320

Sylvia Ghattas 55-1211

Farah Waleed 55-0724

After creating table with no data:

Lock type : Relation Mode :AccessShareLock

Access share lock is a lock mode that allows multiple select queries which is reading of data from the table.This lock is a shared lock which means that more than one session can do it at the same time without errors to any of them. Hence,it works with other access share locks but it does not work with exclusive locks.

Postgre sql uses this lock before inserting data to ensure queries are executed in a correct way.Hence, it will ensure consistency of data when more than one query are trying to access the same table at the same time.

Lock type : Virtualxid Mode :Exclusive Lock

This lock relates to a transaction's virtual transaction ID, it signifies that the lock is exclusive, only one transaction can be held at a time.

- The Virtualxid lock type is a high-level transactional lock applied to the entire schema creation process. This ensures the creation acts as a single unit. If any part of the creation process is inconsistent, the entire operation will fail. In other words, it guarantees success or failure for the entire schema creation.

- PostgreSQL uses exclusive locks when creating any schema, even if empty. This is because it guarantees only one process can create the schema at a time, preventing inconsistencies. Additionally, it ensures that any future modifications will be consistent with the initial schema definition.

	locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid text	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastpat boolea
1	relation	17595	12073	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5/220340	32932	AccessShareLock	true	true
2	virtualxid	[null]	[null]	[null]	[null]	5/220340	[null]	[null]	[null]	[null]	5/220340	32932	ExclusiveLock	true	true

While filing table with data :

	locktype text	database oid	relation oid	page integer	tuple smallint	virtualxid text	transactionid xid	classid oid	objid oid	objsubid smallint	virtualtransaction text	pid integer	mode text	granted boolean	fastp bool
1	relation	17595	12073	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5/220350	32932	AccessShareLock	true	true
2	virtualxid	[null]	[null]	[null]	[null]	5/220350	[null]	[null]	[null]	[null]	5/220350	32932	ExclusiveLock	true	true
3	relation	17595	17608	[null]	[null]	[null]	[null]	[null]	[null]	[null]	10/47289	33164	RowShareLock	true	true
4	relation	17595	17603	[null]	[null]	[null]	[null]	[null]	[null]	[null]	10/47289	33164	RowShareLock	true	true
5	relation	17595	17601	[null]	[null]	[null]	[null]	[null]	[null]	[null]	10/47289	33164	RowShareLock	true	true
6	relation	17595	17596	[null]	[null]	[null]	[null]	[null]	[null]	[null]	10/47289	33164	RowShareLock	true	true
7	relation	17595	17631	[null]	[null]	[null]	[null]	[null]	[null]	[null]	10/47289	33164	RowExclusiveLock	true	true
8	virtualxid	[null]	[null]	[null]	[null]	10/47289	[null]	[null]	[null]	[null]	10/47289	33164	ExclusiveLock	true	true
9	transactionid	[null]	[null]	[null]	[null]	[null]	808165	[null]	[null]	[null]	10/47289	33164	ExclusiveLock	true	fals

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	17595	12073	NULL	NULL	NULL	NULL	NULL	NULL	NULL	5/220350	32932	AccessShareLock	TRUE	TRUE	NULL
virtualxid	NULL	NULL	NULL	NULL	5/220350	NULL	NULL	NULL	NULL	5/220350	32932	ExclusiveLock	TRUE	TRUE	NULL
relation	17595	17608	NULL	NULL	NULL	NULL	NULL	NULL	NULL	10/47289	33164	RowShareLock	TRUE	TRUE	NULL
relation	17595	17603	NULL	NULL	NULL	NULL	NULL	NULL	NULL	10/47289	33164	RowShareLock	TRUE	TRUE	NULL
relation	17595	17601	NULL	NULL	NULL	NULL	NULL	NULL	NULL	10/47289	33164	RowShareLock	TRUE	TRUE	NULL
relation	17595	17596	NULL	NULL	NULL	NULL	NULL	NULL	NULL	10/47289	33164	RowShareLock	TRUE	TRUE	NULL
relation	17595	17631	NULL	NULL	NULL	NULL	NULL	NULL	NULL	10/47289	33164	RowExclusiveLock	TRUE	TRUE	NULL
virtualxid	NULL	NULL	NULL	NULL	10/47289	NULL	NULL	NULL	NULL	10/47289	33164	ExclusiveLock	TRUE	TRUE	NULL
transactionid	NULL	NULL	NULL	NULL	NULL	808165	NULL	NULL	NULL	10/47289	33164	ExclusiveLock	TRUE	FALSE	NULL

Lock type : Relation Mode :AccessShareLock :

It is used here to ensure multiple queries can read (Select) from the table at the same time while the table is getting filled with data. However , no other query can write/modify .

Lock type: Relation Mode: RowShareLock

This lock mode allows several sessions to read the same row simultaneously but prevents any session from updating or deleting the locked row until the lock is released, it is a type of lock acquired on individual rows within the table. The RowShareLock was used to maintain data consistency and data integrity, and because we are using PostgreSQL a more restrictive lock like RowShareLock was used to prevent anomalies, and also in order to avoid data corruption, as PostgreSQL obtain locks on the affected rows, and also to maintain data consistency during bulk insertions.

Lock type: Relation Mode: RowExclusiveLock

PostgreSQL uses this lock is used due to unique constraints being present in the table where data is being inserted, it ensures that no two rows have identical values in the in their respective columns. In case a new row is being inserted with the same

value as another row, postgre uses the RowExclusiveLock to lock the rows with identical values. This stops other transactions from inserting the same duplicate data while the system checks if your insert is allowed. Another reason when RowExclusiveLocks on the referenced rows may be momentarily acquired by PostgreSQL when you insert data that refers to rows that already exist in another table via a foreign key constraint.

Lock type : Virtualxid Mode :Exclusive Lock

This lock remains the same as before (when the schema was empty), it is related/associated with a transaction virtual id. However the reason for postgre using it in that situation differs.

- Here it has a benefit of maintaining data integrity by ensuring the process to be modified at a time. Thus preventing inconsistencies, meaning if 2 insertions are being inserted with the same id this exclusive lock will only allow one insertion to go through.

- Another benefit/reason is to guarantee serializability, meaning that outcomes appear in a sequential way even if they happen in parallel. This is thanks to the locking data during the filling process.

Lock type : transactionid Mode :Exclusive Lock

The mode of this lock remains the same but the type here differs it means that this lock is tied to a specific current transaction, Thus, no other transaction can interfere with the current filling process with this specific transaction id. This can be done for many reasons, including:

- Having a temporary lock that last only for the duration of the data filling within a single transaction
- Also, it has potential for row-level locking, meaning it is not a way to ensure schema-level consistency like when lock type is Virtualxid but it a more detailed and granular control during data insertion.

However, it is mentioned that this (exclusive lock on type transactionid) could show up due to a custom script

After filling in the tables with data:

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	17595	12073	NULL	NULL	NULL	NULL	NULL	NULL	NULL	5/220355	32932	AccessShareLock	TRUE	TRUE	NULL
virtualxid	NULL	NULL	NULL	NULL	5/220355	NULL	NULL	NULL	NULL	5/220355	32932	ExclusiveLock	TRUE	TRUE	NULL

Lock type : Relation Mode: AccessShareLock:

This lock could have been acquired so that PostgreSQL can control simultaneous access to the table, also in order to ensure the read consistency of the table as implementing AccessShareLock can prevent being blocked by concurrent operations while reading. It also prevents data inconsistency by ensuring data read in transactions is consistent and not interrupted in PostgreSQL. This lock mode allows several transactions to read from the tables concurrently but prevents write operations at the same time that could interrupt the ongoing reads.

Lock type: virtualxid Mode: ExclusiveLock:

The explanation doesn't differ much as explained above in different scenarios. Although you'd expect that locks would be released after the tables are completely filled with data but that isn't the case most of the time. The following reasons for it to stay after data is filed are below:

- The transaction may not be simply done yet maybe other actions will be done later on like creating indexes and to remain consistent only one process should be able to do that. Especially that creation of indexes can be delayed by postgre untill data committed.

- Another reason would be if the data loading is too huge and takes time but this is simply a configuration in settings or if the transaction wasnt committed after data insertion is done.

Lock type: Relation Mode: RowExclusiveLock

RowExclusiveLocks may be used internally by PostgreSQL in cleanup or background maintenance procedures. These tasks could include recovering space that was not used, restoring indexes, or rearranging data. These actions may obtain temporary locks on table rows, but they shouldn't cause any disruption.

(while inserting data)

```
"relation"      "17595"      "12073"
        "5/220350"      32932   "AccessShareLock"       true    true
"virtualxid"                                    "5/220350"
"5/220350"      32932   "ExclusiveLock"         true    true
"relation"      "17595"      "17608"
        "10/47289"      33164   "RowShareLock"          true    true
"relation"      "17595"      "17603"
        "10/47289"      33164   "RowShareLock"          true    true
"relation"      "17595"      "17601"
        "10/47289"      33164   "RowShareLock"          true    true
"relation"      "17595"      "17596"
        "10/47289"      33164   "RowShareLock"          true    true
"relation"      "17595"      "17631"
        "10/47289"      33164   "RowExclusiveLock"      true    true
"virtualxid"                                    "10/47289"
"10/47289"      33164   "ExclusiveLock"         true    true
"transactionid"                                 "808165"
"10/47289"      33164   "ExclusiveLock"         true    false
```

(after inserting data)

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	17595	12073	NULL	NULL	NULL	NULL	NULL	NULL	NULL	5/220355	32932	AccessShareLock	TRUE	TRUE	NULL
virtualxid	NULL	NULL	NULL	NULL	5/220355	NULL	NULL	NULL	NULL	5/220355	32932	ExclusiveLock	TRUE	TRUE	NULL

Sample results of query 1:

Data Output | Messages | Notifications

	id integer	name character varying (20)	tot_credit integer	department character varying (20)	advisor_id integer	student_id integer	section_id integer	grade real	section_id integer	semester integer	year integer	instructor_id integer
47	47	Student 47	47	CSEN	3	47	46	1	46	1	2024	47
48	48	Student 48	48	CSEN	4	48	47	1.3	47	1	2024	48
49	49	Student 49	49	CSEN	5	49	48	1.6	48	1	2024	49
50	50	Student 50	50	CSEN	6	50	49	1.9	49	1	2024	50
51	51	Student 51	51	CSEN	7	51	50	2.2	50	1	2024	51
52	52	Student 52	52	CSEN	8	52	51	2.5	51	1	2024	52
53	53	Student 53	53	CSEN	9	53	52	2.8	52	1	2024	53
54	54	Student 54	54	CSEN	10	54	53	3.1	53	1	2024	54
55	55	Student 55	55	CSEN	11	55	54	3.4	54	1	2024	55
56	56	Student 56	56	CSEN	12	56	55	3.7	55	1	2024	56
57	57	Student 57	57	CSEN	13	57	56	4	56	1	2024	57
58	58	Student 58	58	CSEN	14	58	57	4.3	57	1	2024	58
59	59	Student 59	59	CSEN	15	59	58	4.6	58	1	2024	59
60	60	Student 60	60	CSEN	1	60	59	4.9	59	1	2024	60

Total rows: 1000 of 1199 | Query complete 00:00:00.138 | Ln 9, Col 1

Query 1 inner query:

	student_id integer	section_id integer	grade real	section_id integer	semester integer	year integer	instructor_id integer	course_id integer	classroom_building integer	classroom_room_no integer
1	31	30	0.7	30	1	2024	31	31	31	31
2	32	31	1	31	1	2024	32	32	32	32
3	33	32	1.3	32	1	2024	33	33	33	33
4	34	33	1.6	33	1	2024	34	34	34	34
5	35	34	1.9	34	1	2024	35	35	35	35
6	36	35	2.2	35	1	2024	36	36	36	36
7	37	36	2.5	36	1	2024	37	37	37	37
8	38	37	2.8	37	1	2024	38	38	38	38
9	39	38	3.1	38	1	2024	39	39	39	39
10	40	39	3.4	39	1	2024	40	40	40	40
11	41	40	3.7	40	1	2024	41	41	41	41
12	42	41	4	41	1	2024	42	42	42	42
13	43	42	4.3	42	1	2024	43	43	43	43
14	44	43	4.6	43	1	2024	44	44	44	44
15	45	44	4.9	44	1	2024	45	45	45	45

Total rows: 404 of 404 | Query complete 00:00:00.166

Analysis: Schema 1:

Without indices:

Code:

```
set enable_indexscan=off;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=on;

Explain Analyze

select * from (select * from student

        where department = 'CSEN') as CS1_student

full outer join

(select * from takes t inner join section s

        on t.section_id = s.section_id

                where semester = 1 and  year = 2024) as sem1_student

                on CS1_student.id = sem1_student.student_id;
```

	QUERY PLAN text
1	Hash Full Join (cost=10000005031.66..10000006534.34 rows=1161 width=71) (actual time=18.795..23.248 rows=119...
2	Hash Cond: (student.id = t.student_id)
3	-> Seq Scan on student (cost=0.00..1498.25 rows=1161 width=31) (actual time=0.026..4.304 rows=1199 loops=1)
4	Filter: ((department)::text = 'CSEN'::text)
5	Rows Removed by Filter: 70741
6	-> Hash (cost=5026.45..5026.45 rows=417 width=40) (actual time=18.755..18.757 rows=404 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 37kB
8	-> Nested Loop (cost=0.00..5026.45 rows=417 width=40) (actual time=0.026..18.666 rows=404 loops=1)
9	Join Filter: (t.section_id = s.section_id)
10	Rows Removed by Join Filter: 251600
11	-> Seq Scan on takes t (cost=0.00..12.70 rows=770 width=12) (actual time=0.009..0.065 rows=770 loops=1)
12	-> Materialize (cost=0.00..20.15 rows=433 width=28) (actual time=0.000..0.010 rows=327 loops=770)
13	-> Seq Scan on section s (cost=0.00..17.98 rows=433 width=28) (actual time=0.006..0.058 rows=433 loop...
14	Filter: ((semester = 1) AND (year = 2024))
15	Rows Removed by Filter: 366
16	Planning Time: 3.057 ms
17	Execution Time: 23.318 ms

Query Without an Index

When the query was executed without an index, the database performed a full sequential scan on both the student and course sections tables. It used nested loop joins to match students with the matching course sections,which lead to a high cost and inefficiency, since sequential scans and nested loops are amongst the most costly things to perform on a database.

After BTree index:

Some PKeys were dropped to ensure sequential scan would be done for columns that are not indexed

Index name: btreedep / table: student / column: dep_name

Code:

Create index Btreedep on student(department);

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;

	QUERY PLAN text
1	Hash Full Join (cost=30000005038.42..30000006656.13 rows=1161 width=71) (actual time=20.330..20.578 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Index Scan using btreedep on student (cost=0.29..1613.58 rows=1161 width=31) (actual time=0.030..0.109 rows=1199 loops=1)
4	Index Cond: ((department)::text = 'CSEN'::text)
5	-> Hash (cost=20000005032.92..20000005032.92 rows=417 width=40) (actual time=20.291..20.292 rows=404 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 37kB
7	-> Nested Loop (cost=20000000000.00..20000005032.92 rows=417 width=40) (actual time=0.023..20.225 rows=404 loops=1)
8	Join Filter: (t.section_id = s.section_id)
9	Rows Removed by Join Filter: 333006
10	-> Seq Scan on takes t (cost=10000000000.00..10000000012.70 rows=770 width=12) (actual time=0.009..0.058 rows=770 loops=1)
11	-> Materialize (cost=10000000000.00..10000000020.15 rows=433 width=28) (actual time=0.000..0.011 rows=433 loops=770)
12	-> Seq Scan on section s (cost=10000000000.00..10000000017.99 rows=433 width=28) (actual time=0.005..0.049 rows=433 loop...
13	Filter: ((semester = 1) AND (year = 2024))
14	Rows Removed by Filter: 366
15	Planning Time: 0.130 ms
16	Execution Time: 20.651 ms

Query with B+ Trees Indices Only

Here, the query used B+ tree index on the department column of the student table. It performs index scans followed by nested loop joins.

The estimated cost is significantly lower than without the index. This is because index scans are faster than full sequential scans, reducing the number of rows to be read and processed. B+ tree indices provide efficient retrieval of records by allowing the database engine to quickly locate rows that meet the specified conditions, hence, reducing the I/O operations.

Using Hash index:

Index name: trialIndex / table: Student / column: department

Before disabling pkeys to disable unwanted indices:

Code:

```
create index trialIndex on student using hash("department");

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=on;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Hash Full Join (cost=102.99..2515.73 rows=1161 width=71) (actual time=0.314..0.610 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Index Scan using trialindex on student (cost=0.00..2408.32 rows=1161 width=31) (actual time=0.006..0.151 rows=1199 loops=1)
4	Index Cond: ((department)::text = 'CSEN'::text)
5	-> Hash (cost=97.77..97.77 rows=417 width=40) (actual time=0.302..0.302 rows=404 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 37kB
7	-> Hash Join (cost=50.94..97.77 rows=417 width=40) (actual time=0.124..0.255 rows=404 loops=1)
8	Hash Cond: (t.section_id = s.section_id)
9	-> Index Scan using takes_pkey on takes t (cost=0.28..45.08 rows=770 width=12) (actual time=0.013..0.067 rows=770 loops=1)
10	-> Hash (cost=45.26..45.26 rows=433 width=28) (actual time=0.106..0.106 rows=433 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 34kB
12	-> Index Scan using section_pkey on section s (cost=0.28..45.26 rows=433 width=28) (actual time=0.007..0.077 rows=433 loop...
13	Filter: ((semester = 1) AND (year = 2024))
14	Rows Removed by Filter: 366
15	Planning Time: 0.137 ms
16	Execution Time: 0.658 ms

After disabling pkeys to disable unwanted indices:

	QUERY PLAN text
1	Hash Full Join (cost=20000000048.37..20000002461.11 rows=1161 width=71) (actual time=0.432..0.832 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Index Scan using trialindex on student (cost=0.00..2408.32 rows=1161 width=31) (actual time=0.012..0.210 rows=1199 loops=1)
4	Index Cond: ((department)::text = 'CSEN'::text)
5	-> Hash (cost=20000000043.15..20000000043.15 rows=417 width=40) (actual time=0.411..0.412 rows=404 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 37kB
7	-> Hash Join (cost=20000000023.40..20000000043.15 rows=417 width=40) (actual time=0.153..0.335 rows=404 loops=1)
8	Hash Cond: (t.section_id = s.section_id)
9	-> Seq Scan on takes t (cost=10000000000.00..10000000012.70 rows=770 width=12) (actual time=0.017..0.082 rows=770 loops=1)
10	-> Hash (cost=10000000017.99..10000000017.99 rows=433 width=28) (actual time=0.125..0.126 rows=433 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 34kB
12	-> Seq Scan on section s (cost=10000000000.00..10000000017.99 rows=433 width=28) (actual time=0.010..0.081 rows=433 loop...
13	Filter: ((semester = 1) AND (year = 2024))
14	Rows Removed by Filter: 366
15	Planning Time: 0.748 ms
16	Execution Time: 0.906 ms

Query with Hash Indices Only

Here, hash index is used on the department column in table student. The query execution plan involves hash index scans and nested loop joins. The cost is lower than in scenario 1 but higher compared to B+ tree indexes. This is due to the limited use case of hash indexes for equality searches and the overhead of maintaining hash tables.

Hash indexes are optimal for equality comparisons but less efficient for range scans and sorting, which can increase the cost in some query scenarios.

Creating BRIN index:

Index type: BRIN / index name: trialbrin / column: section_id / table: takes

set enable_indexscan=off;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;

bitmapscan is enabled to allow for the BRIN index to show

	QUERY PLAN text
1	Hash Full Join (cost=30002255809.12..30002257311.79 rows=1161 width=71) (actual time=17.857..21.709 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Seq Scan on student (cost=10000000000.00..10000001498.25 rows=1161 width=31) (actual time=0.011..3.708 rows=1199 loops=1)
4	Filter: ((department)::text = 'CSEN'::text)
5	Rows Removed by Filter: 70741
6	-> Hash (cost=10002255803.90..10002255803.90 rows=417 width=40) (actual time=17.838..17.841 rows=404 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 37kB
8	-> Nested Loop (cost=10000005196.03..10002255803.90 rows=417 width=40) (actual time=0.130..17.748 rows=404 loops=1)
9	-> Seq Scan on section s (cost=10000000000.00..10000000017.99 rows=433 width=28) (actual time=0.007..0.079 rows=433 loop...
10	Filter: ((semester = 1) AND (year = 2024))
11	Rows Removed by Filter: 366
12	-> Bitmap Heap Scan on takes t (cost=5196.03..5209.66 rows=1 width=12) (actual time=0.018..0.039 rows=1 loops=433)
13	Recheck Cond: (section_id = s.section_id)
14	Rows Removed by Index Recheck: 717
15	Heap Blocks: lossy=2020
16	-> Bitmap Index Scan on trialbrin (cost=0.00..5196.03 rows=770 width=0) (actual time=0.004..0.004 rows=47 loops=433)
17	Index Cond: (section_id = s.section_id)
18	Planning Time: 0.132 ms
19	Execution Time: 21.976 ms

Scenario 4: Query with BRIN Indices Only

Here, the query uses BRIN index on the section_id column in the takes table. The execution involves BRIN index scans and joins. The cost is reduced compared to the no-index scenario but generally higher than B+ tree indexes. This is due to the nature of BRIN indices and the fact that they work best with ranges.

BRIN indexes are efficient for very large tables where data is naturally clustered, but they are less precise than B+ trees, leading to additional filtering steps.

Using GIN index:

index name: GinIndex / column: department / table: student

CREATE INDEX GinIndex ON student USING GIN (to_tsvector('english', department)) WHERE department IN ('CSEN');

set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;

	QUERY PLAN text
1	Hash Full Join (cost=30000005046.52..30000005668.80 rows=1161 width=71) (actual time=27.362..27.589 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Bitmap Heap Scan on student (cost=8.39..626.25 rows=1161 width=31) (actual time=0.068..0.125 rows=1199 loops=1)
4	Recheck Cond: ((department)::text = 'CSEN'::text)
5	Heap Blocks: exact=10
6	-> Bitmap Index Scan on ginindex (cost=0.00..8.10 rows=1161 width=0) (actual time=0.060..0.060 rows=1199 loops=1)
7	-> Hash (cost=20000005032.92..20000005032.92 rows=417 width=40) (actual time=27.286..27.286 rows=404 loops=1)
8	Buckets: 1024 Batches: 1 Memory Usage: 37kB
9	-> Nested Loop (cost=20000000000.00..20000005032.92 rows=417 width=40) (actual time=0.027..27.050 rows=404 loops=1)
10	Join Filter: (t.section_id = s.section_id)
11	Rows Removed by Join Filter: 333006
12	-> Seq Scan on takes t (cost=10000000000.00..10000000012.70 rows=770 width=12) (actual time=0.006..0.108 rows=770 loops=1)
13	-> Materialize (cost=10000000000.00..10000000020.15 rows=433 width=28) (actual time=0.000..0.014 rows=433 loops=770)
14	-> Seq Scan on section s (cost=10000000000.00..10000000017.99 rows=433 width=28) (actual time=0.005..0.047 rows=433 loop...
15	Filter: ((semester = 1) AND (year = 2024))
16	Rows Removed by Filter: 366
17	Planning Time: 1.948 ms
18	Execution Time: 27.675 ms

Query with GIN Index

GIN index was applicable here because there was a complex data type that was being searched on, which is the department name, and GIN indices can work well with reducing the cost and increasing the efficiency of complex data types as such.

Using indices on all columns:

Indices created:

Index type: BTree / name: Btreedep/ column: department / table: student

Index type: BRIN / name: YearSemBrin / column: year , semester / table: section

Index type: Hash / name: HashTakes/ column: section_id / table: takes

Index type: Hash / name: hashSec / column: section_id / table: section

Code:

```
create index YearSemBrin on "section" using brin(year, semester);

Create index Btreedep on student(department);

create index hashTakes on "takes" using Hash("section_id");

create index hashSec on "section" using Hash("section_id");


set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Hash Full Join (cost=10000000108.54..10000000730.82 rows=1161 width=71) (actual time=0.402..0.623 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Bitmap Heap Scan on student (cost=17.29..635.15 rows=1161 width=31) (actual time=0.034..0.095 rows=1199 loops=1)
4	Recheck Cond: ((department)::text = 'CSEN'::text)
5	Heap Blocks: exact=10
6	-> Bitmap Index Scan on btreedep (cost=0.00..17.00 rows=1161 width=0) (actual time=0.029..0.029 rows=1199 loops=1)
7	Index Cond: ((department)::text = 'CSEN'::text)
8	-> Hash (cost=86.03..86.03 rows=417 width=40) (actual time=0.363..0.363 rows=404 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 37kB
10	-> Nested Loop (cost=12.14..86.03 rows=417 width=40) (actual time=0.038..0.323 rows=404 loops=1)
11	-> Bitmap Heap Scan on section s (cost=12.14..30.13 rows=433 width=28) (actual time=0.015..0.081 rows=433 loops=1)
12	Recheck Cond: ((year = 2024) AND (semester = 1))
13	Rows Removed by Index Recheck: 366
14	Heap Blocks: lossy=6
15	-> Bitmap Index Scan on yearsembrin (cost=0.00..12.03 rows=799 width=0) (actual time=0.010..0.010 rows=60 loop...
16	Index Cond: ((year = 2024) AND (semester = 1))
17	-> Index Scan using hashtakes on takes t (cost=0.00..0.12 rows=1 width=12) (actual time=0.000..0.000 rows=1 loops=4...
18	Index Cond: (section_id = s.section_id)
19	Planning Time: 3.413 ms
20	Execution Time: 0.681 ms

Query with Mixed Indices on All Columns

This is a more complex scenario. The query uses a combination of B+ tree, hash, BRIN, and GIN indexes on relevant columns. The planner selects the optimal indexes for scanning and joining tables.

The cost is lower due to the optimal use of different indexes for various parts of the query, balancing the strengths of each index type. Using multiple index types allows the query planner to choose the most efficient index for each operation, enhancing performance and reducing overall execution cost.

Combination of indices to optimize performance:

Indices created and why:

Name: BtreeSemester / Type: BTree/ Column: semester / Table: section

BTrees work best with precise data that can be compared, since we want semester 1, the BTree will read the data under the node stored at semester 1 hence improving performance, it was not chosen for year since years inserted only included 2024

Name: hashSection / Type: hash / Column: section_id / Table: takes

Hash indices work best with precise values, and takes and section are joined based on section_id hence it makes sense to optimize the join using a hash index

Name: hashDep / Type: hash / Column: department / Table: student

As explained, hash indices work best with precise values, and the query was looking for "department = CSEN" which is a precise value hence why it was used here

Code:

```
--create index BtreeSemester on section(semester);

-- create index hashSection on takes(section_id);

-- create index hashDep on "student" using Hash("department");

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=on;

set enable_seqscan=off;
```

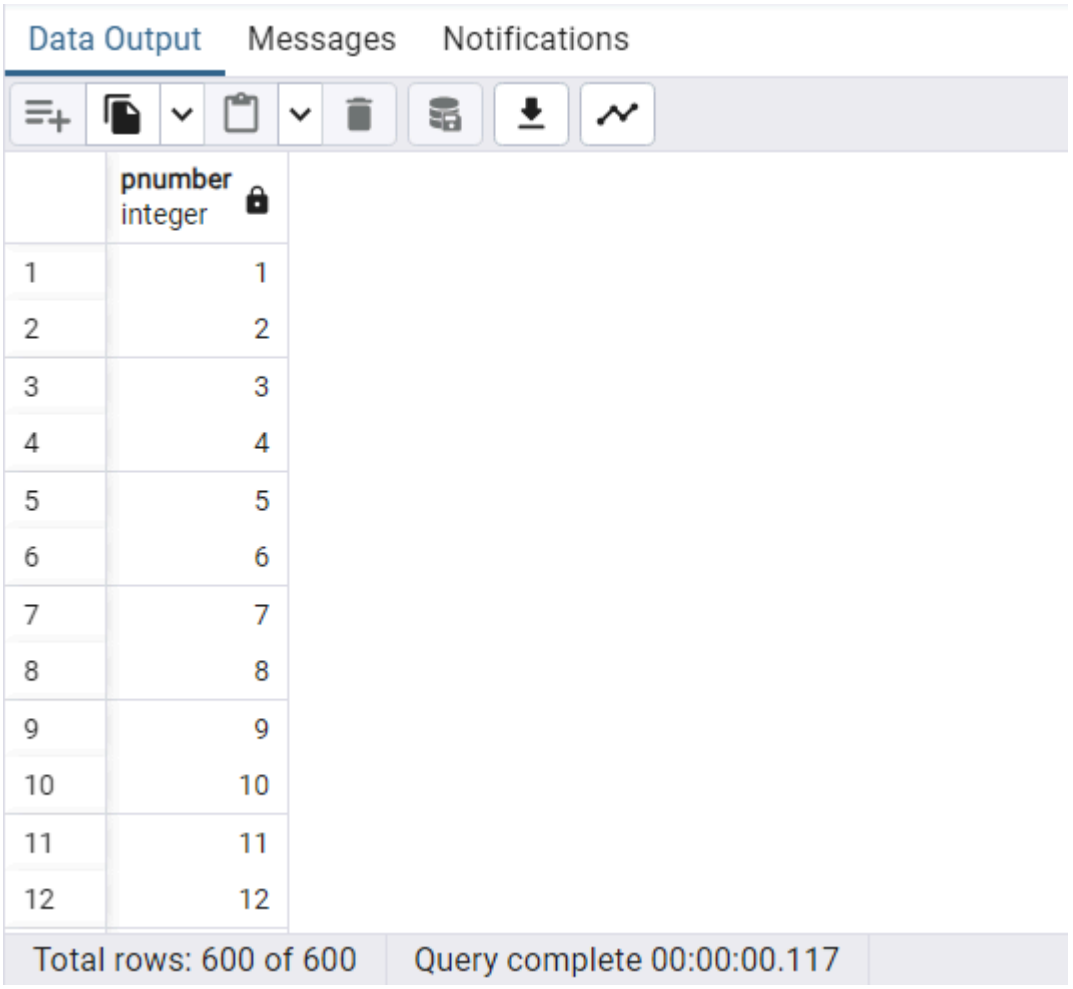
BRIN was not used since BRIN works best with ranges of values and there are no ranges queried here.

As seen the cost of the execution significantly decreased compared to all other executions prior to this one

	QUERY PLAN text
1	Hash Full Join (cost=87.90..2500.64 rows=1161 width=71) (actual time=0.525..0.989 rows=1199 loops=1)
2	Hash Cond: (student.id = t.student_id)
3	-> Index Scan using hashdep on student (cost=0.00..2408.32 rows=1161 width=31) (actual time=0.012..0.238 rows=1199 loops=1)
4	Index Cond: ((department)::text = 'CSEN'::text)
5	-> Hash (cost=82.68..82.68 rows=417 width=40) (actual time=0.504..0.507 rows=404 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 37kB
7	-> Hash Join (cost=36.08..82.68 rows=417 width=40) (actual time=0.166..0.425 rows=404 loops=1)
8	Hash Cond: (t.section_id = s.section_id)
9	-> Index Scan using hashsection on takes t (cost=0.28..39.83 rows=770 width=12) (actual time=0.041..0.158 rows=770 loops=1)
10	-> Hash (cost=30.39..30.39 rows=433 width=28) (actual time=0.117..0.118 rows=433 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 34kB
12	-> Index Scan using btreesemester on section s (cost=0.15..30.39 rows=433 width=28) (actual time=0.010..0.075 rows=433 loop...
13	Index Cond: (semester = 1)
14	Filter: (year = 2024)
15	Planning Time: 1.281 ms
16	Execution Time: 1.078 ms

Schema 2 :

Query 2 sample results:



	pnumber integer
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12

Total rows: 600 of 600 Query complete 00:00:00.117

Query 2

Without indices:

Code:

```
set enable_indexscan=off;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=on;


explain analyze

select distinct pnumber

                from project where pnumber in

                (select pnumber from project, department d, employee e

                where e.dno=d.dnumber and d.mgr_snn=ssn

                and  e.lname='employee1' and project.pnumber=d.dnumber)

        or

                pnumber in (select pno from works_on, employee

                 where essn=ssn and lname='employee1');
```

	QUERY PLAN text
1	HashAggregate (cost=8542.68..8611.68 rows=6900 width=4) (actual time=26.523..26.593 rows=600 loops=1)
2	Group Key: project.pnumber
3	Batches: 1 Memory Usage: 241kB
4	-> Seq Scan on project (cost=8255.43..8525.43 rows=6900 width=4) (actual time=24.873..26.357 rows=600 loops=1)
5	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
6	Rows Removed by Filter: 8600
7	SubPlan 1
8	-> Nested Loop (cost=0.00..2380.88 rows=1 width=4) (actual time=0.111..10.241 rows=1 loops=1)
9	Join Filter: (e.dno = project_1.pnumber)
10	Rows Removed by Join Filter: 4
11	-> Nested Loop (cost=0.00..2041.88 rows=1 width=8) (actual time=0.107..10.236 rows=1 loops=1)
12	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
13	Rows Removed by Join Filter: 89854
14	-> Seq Scan on employee e (cost=0.00..463.00 rows=600 width=8) (actual time=0.006..1.785 rows=600 loops=1)
15	Filter: (lname = 'employee1'::bpchar)
16	Rows Removed by Filter: 15400
17	-> Materialize (cost=0.00..4.25 rows=150 width=8) (actual time=0.000..0.005 rows=150 loops=600)
18	-> Seq Scan on department d (cost=0.00..3.50 rows=150 width=8) (actual time=0.006..0.018 rows=150 loop...
19	-> Seq Scan on project project_1 (cost=0.00..224.00 rows=9200 width=4) (actual time=0.003..0.003 rows=5 loops=...
20	SubPlan 2
21	-> Nested Loop (cost=0.00..5874.50 rows=22 width=4) (actual time=0.032..14.475 rows=600 loops=1)
22	Join Filter: (works_on.essn = employee.ssn)
23	Rows Removed by Join Filter: 179700
24	-> Seq Scan on works_on (cost=0.00..10.00 rows=600 width=8) (actual time=0.020..0.058 rows=600 loops=1)
25	-> Materialize (cost=0.00..466.00 rows=600 width=4) (actual time=0.000..0.010 rows=300 loops=600)
26	-> Seq Scan on employee (cost=0.00..463.00 rows=600 width=4) (actual time=0.006..0.105 rows=600 loops=1)
27	Filter: (lname = 'employee1'::bpchar)
28	Planning Time: 0.354 ms
29	Execution Time: 26.791 ms

Scenario 1: Query Without an Index

Execution Plan of the Query: The query involves full table scans of the project, department, employee, and works_on tables. It uses nested loop joins and filters to evaluate the subqueries and the main query.

Estimated Cost of the Plan: The estimated cost is very high due to multiple full table scans and joins, which are resource-intensive operations.

Why? Without indexes, the database must read each row of the tables and perform many comparisons to satisfy the query conditions, resulting in inefficient execution.

After BTree index:

Some PKeys were dropped to ensure sequential scan would be done for columns that are not indexed

Index name: btreeIndex / table: employee / column: ssn

Code:

```
--Create index BtreeIndex on employee(ssn);

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Unique (cost=40000003181.15..40000003215.65 rows=6900 width=4) (actual time=5.800..5.935 rows=600 loops=1)
2	-> Sort (cost=40000003181.15..40000003198.40 rows=6900 width=4) (actual time=5.799..5.836 rows=600 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 25kB
5	-> Seq Scan on project (cost=40000002471.19..40000002741.19 rows=6900 width=4) (actual time=3.510..5.755 rows=600 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	Rows Removed by Filter: 8600
8	SubPlan 1
9	-> Nested Loop (cost=20000000000.28..20000001042.12 rows=1 width=4) (actual time=0.048..2.104 rows=1 loops=1)
10	Join Filter: (e.dno = project_1.pnumber)
11	Rows Removed by Join Filter: 9199
12	-> Nested Loop (cost=10000000000.28..10000000703.12 rows=1 width=8) (actual time=0.040..0.308 rows=1 loops=1)
13	-> Seq Scan on department d (cost=10000000000.00..10000000003.50 rows=150 width=8) (actual time=0.011..0.025 rows=150 loops=...
14	-> Index Scan using btreeindex on employee e (cost=0.29..4.65 rows=1 width=8) (actual time=0.002..0.002 rows=0 loops=150)
15	Index Cond: (ssn = d.mgr_snn)
16	Filter: ((lname = 'employee1'::bpchar) AND (d.dnumber = dno))
17	Rows Removed by Filter: 1
18	-> Seq Scan on project project_1 (cost=10000000000.00..10000000224.00 rows=9200 width=4) (actual time=0.005..0.888 rows=9200 loop...
19	SubPlan 2
20	-> Nested Loop (cost=10000000000.28..10000001429.01 rows=22 width=4) (actual time=0.029..1.185 rows=600 loops=1)
21	-> Seq Scan on works_on (cost=10000000000.00..10000000010.00 rows=600 width=8) (actual time=0.020..0.100 rows=600 loops=1)
22	-> Index Scan using btreeindex on employee (cost=0.29..2.37 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=600)
23	Index Cond: (ssn = works_on.essn)
24	Filter: (lname = 'employee1'::bpchar)
25	Planning Time: 1.770 ms
26	Execution Time: 6.173 ms

Scenario 2: Query with B+ Trees Indices Only

Execution Plan of the Query: B+ tree indexes are created on pnumber in project, dno in employee, dnumber in department, mgr_snn in department, ssn in employee, and lname in employee. The query uses index scans instead of full table scans.

Estimated Cost of the Plan: The cost is significantly lower than scenario 1 because index scans are much faster, reducing the number of rows processed in each table.

Why? B+ tree indexes facilitate efficient row retrieval for equality and range conditions, significantly speeding up the query by reducing I/O operations and processing time.

Using Hash index:

Index name: trialIndex / table: employee / column: lname

Index name: HashIndex / table: employee / column: ssn

Before disabling pkeys to disable unwanted indices:

Code:

```
create index trialIndex on employee using hash("lname");

create index hashIndex on employee using hash("ssn");

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=on;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Unique (cost=40000003133.39..40000003167.89 rows=6900 width=4) (actual time=9.755..9.993 rows=600 loops=1)
2	-> Sort (cost=40000003133.39..40000003150.64 rows=6900 width=4) (actual time=9.752..9.809 rows=600 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 25kB
5	-> Seq Scan on project (cost=40000002423.43..40000002693.43 rows=6900 width=4) (actual time=6.065..9.688 rows=600 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	Rows Removed by Filter: 8600
8	SubPlan 1
9	-> Nested Loop (cost=20000000000.00..20000001079.38 rows=1 width=4) (actual time=0.115..3.403 rows=1 loops=1)
10	Join Filter: (e.dno = project_1.pnumber)
11	Rows Removed by Join Filter: 9199
12	-> Nested Loop (cost=10000000000.00..10000000740.38 rows=1 width=8) (actual time=0.087..0.550 rows=1 loops=1)
13	-> Seq Scan on department d (cost=10000000000.00..10000000003.50 rows=150 width=8) (actual time=0.014..0.034 rows=150 loops=...
14	-> Index Scan using hashindex on employee e (cost=0.00..4.90 rows=1 width=8) (actual time=0.003..0.003 rows=0 loops=150)
15	Index Cond: (ssn = d.mgr_snn)
16	Filter: ((lname = 'employee1'::bpchar) AND (d.dnumber = dno))
17	Rows Removed by Filter: 1
18	-> Seq Scan on project project_1 (cost=10000000000.00..10000000224.00 rows=9200 width=4) (actual time=0.023..1.179 rows=9200 loop...
19	SubPlan 2
20	-> Nested Loop (cost=10000000000.00..10000001344.00 rows=22 width=4) (actual time=0.064..2.214 rows=600 loops=1)
21	-> Seq Scan on works_on (cost=10000000000.00..10000000010.00 rows=600 width=8) (actual time=0.052..0.192 rows=600 loops=1)
22	-> Index Scan using hashindex on employee (cost=0.00..2.21 rows=1 width=4) (actual time=0.002..0.003 rows=1 loops=600)
23	Index Cond: (ssn = works_on.essn)
24	Filter: (lname = 'employee1'::bpchar)
25	Planning Time: 2.073 ms
26	Execution Time: 10.120 ms

Scenario 3: Query with Hash Indices Only

Execution Plan of the Query: Hash indexes are created on pnumber, dno, dnumber, mgr_snn, ssn, and lname. The query uses hash index scans and nested loop joins.

Estimated Cost of the Plan: The cost is lower than scenario 1 but generally higher compared to B+ tree indexes because hash indexes are less versatile and efficient for range scans and ordering.

Why? Hash indexes are optimized for equality comparisons but are less efficient for operations requiring sorting or range scans, resulting in a moderate reduction in execution cost.

Creating BRIN index:

Index type: BRIN / index name: BrinIndex / column: department / table: dnumber

```
create index BrinIndex on department using brin("dnumber");

set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;
```

bitmapscan is enabled to allow for the BRIN index to show

	QUERY PLAN text
1	Unique (cost=50004331161.86..50004331196.36 rows=6900 width=4) (actual time=119.215..119.510 rows=600 loops=1)
2	-> Sort (cost=50004331161.86..50004331179.11 rows=6900 width=4) (actual time=119.210..119.283 rows=600 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 25kB
5	-> Seq Scan on project (cost=50004330451.91..50004330721.91 rows=6900 width=4) (actual time=114.928..119.122 rows=600 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	Rows Removed by Filter: 8600
8	SubPlan 1
9	-> Nested Loop (cost=20000007200.03..20004324577.35 rows=1 width=4) (actual time=0.227..25.571 rows=1 loops=1)
10	Join Filter: (e.dno = project_1.pnumber)
11	Rows Removed by Join Filter: 9199
12	-> Nested Loop (cost=10000007200.03..10004324238.35 rows=1 width=8) (actual time=0.216..23.668 rows=1 loops=1)
13	-> Seq Scan on employee e (cost=10000000000.00..10000000463.00 rows=600 width=8) (actual time=0.009..2.994 rows=600 loops=1)
14	Filter: (lname = 'employee1'::bpchar)
15	Rows Removed by Filter: 15400
16	-> Bitmap Heap Scan on department d (cost=7200.03..7206.28 rows=1 width=8) (actual time=0.030..0.030 rows=0 loops=600)
17	Recheck Cond: (dnumber = e.dno)
18	Rows Removed by Index Recheck: 149
19	Filter: (e.ssn = mgr_snn)
20	Rows Removed by Filter: 1
21	Heap Blocks: lossy=1200
22	-> Bitmap Index Scan on brinindex (cost=0.00..7200.03 rows=150 width=0) (actual time=0.008..0.008 rows=20 loops=600)
23	Index Cond: (dnumber = e.dno)
24	-> Seq Scan on project project_1 (cost=10000000000.00..10000000224.00 rows=9200 width=4) (actual time=0.007..0.944 rows=9200 loop...
25	SubPlan 2
26	-> Nested Loop (cost=20000000000.00..20000005874.50 rows=22 width=4) (actual time=0.049..88.495 rows=600 loops=1)
27	Join Filter: (works_on.essn = employee.ssn)
28	Rows Removed by Join Filter: 359400
29	-> Seq Scan on works_on (cost=10000000000.00..10000000010.00 rows=600 width=8) (actual time=0.028..0.322 rows=600 loops=1)
30	-> Materialize (cost=10000000000.00..10000000466.00 rows=600 width=4) (actual time=0.000..0.063 rows=600 loops=600)
31	-> Seq Scan on employee (cost=10000000000.00..10000000463.00 rows=600 width=4) (actual time=0.011..3.096 rows=600 loops=1)
32	Filter: (lname = 'employee1'::bpchar)
33	Rows Removed by Filter: 15400
34	Planning Time: 0.517 ms
35	Execution Time: 124.309 ms

Scenario 4: Query with BRIN Indices Only

Execution Plan of the Query: BRIN indexes are created on pnumber, dno, dnumber, mgr_snn, ssn, and lname. The query utilizes BRIN index scans and nested loop joins.

Estimated Cost of the Plan: The cost is reduced compared to scenario 1 but higher than with B+ tree indexes, due to the block-range nature of BRIN indexes, which are less precise.

Why? BRIN indexes are efficient for large tables with naturally clustered data but involve additional filtering steps to locate precise rows, leading to higher cost than B+ trees.

Using GIN index:

Index type: GIN / index name: GinIndex / column: lname / table: employee

```
CREATE INDEX GinIndex ON employee USING GIN (to_tsvector('english', lname))
WHERE lname IN ('employee1');
```

```
set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=on;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Unique (cost=40000008596.91..40000008631.41 rows=6900 width=4) (actual time=68.235..68.528 rows=600 loops=1)
2	-> Sort (cost=40000008596.91..40000008614.16 rows=6900 width=4) (actual time=68.230..68.280 rows=600 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 25kB
5	-> Seq Scan on project (cost=40000007886.96..40000008156.96 rows=6900 width=4) (actual time=65.200..68.178 rows=600 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	Rows Removed by Filter: 8600
8	SubPlan 1
9	-> Nested Loop (cost=20000000008.26..20000002196.64 rows=1 width=4) (actual time=0.219..11.256 rows=1 loops=1)
10	Join Filter: (e.dno = project_1.pnumber)
11	Rows Removed by Join Filter: 9199
12	-> Nested Loop (cost=10000000008.26..10000001857.64 rows=1 width=8) (actual time=0.213..9.720 rows=1 loops=1)
13	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
14	Rows Removed by Join Filter: 89999
15	-> Bitmap Heap Scan on employee e (cost=8.26..278.76 rows=600 width=8) (actual time=0.081..0.146 rows=600 loops=1)
16	Recheck Cond: (lname = 'employee1'::bpchar)
17	Heap Blocks: exact=10
18	-> Bitmap Index Scan on ginindex (cost=0.00..8.11 rows=600 width=0) (actual time=0.073..0.074 rows=600 loops=1)
19	-> Materialize (cost=10000000000.00..10000000004.25 rows=150 width=8) (actual time=0.000..0.007 rows=150 loops=600)
20	-> Seq Scan on department d (cost=10000000000.00..10000000003.50 rows=150 width=8) (actual time=0.008..0.026 rows=150 loop...
21	-> Seq Scan on project project_1 (cost=10000000000.00..10000000224.00 rows=9200 width=4) (actual time=0.004..0.797 rows=9200 loop...
22	SubPlan 2
23	-> Nested Loop (cost=10000000008.26..10000005690.26 rows=22 width=4) (actual time=0.087..53.370 rows=600 loops=1)
24	Join Filter: (works_on.essn = employee.ssn)
25	Rows Removed by Join Filter: 359400
26	-> Seq Scan on works_on (cost=10000000000.00..10000000010.00 rows=600 width=8) (actual time=0.026..0.191 rows=600 loops=1)
27	-> Materialize (cost=8.26..281.76 rows=600 width=4) (actual time=0.000..0.036 rows=600 loops=600)
28	-> Bitmap Heap Scan on employee (cost=8.26..278.76 rows=600 width=4) (actual time=0.053..0.115 rows=600 loops=1)
29	Recheck Cond: (lname = 'employee1'::bpchar)
30	Heap Blocks: exact=10
31	-> Bitmap Index Scan on ginindex (cost=0.00..8.11 rows=600 width=0) (actual time=0.049..0.049 rows=600 loops=1)
32	Planning Time: 0.394 ms
33	Execution Time: 68.725 ms

Scenario 5: Query with GIN Index

Execution Plan of the Query: Assuming GIN indexes are applicable (e.g., for a full-text search on lname), the query uses GIN index scans alongside other appropriate index scans.

Estimated Cost of the Plan: The cost is dependent on the GIN index's use case but generally effective for full-text search operations.

Why? GIN indexes are specialized for indexing composite values or full-text search, providing efficient retrieval for such operations, though not directly enhancing all aspects of this query.

Using indices on all columns:

Indices created:

Index type: BTree / name: IndexSSN/ column: employee / table: ssn

Index type: Hash / name: IndexDnumEmp / column: dno / table: employee

Index type: Hash / name: IndexDnum/ column: dnumber / table: department

Index type: BTree / name: IndexPnum / column: pnumber / table: project

Index type: Hash / name: mgrIndex / column: mgr_snn / table: department

Index type: Hash / name: IndexEssn / column: essn / table: works_on

Index type: gin / name: GinIndex / column: lname / table: employee

Code:

```
create index IndexDnum on department using hash("dnumber");

create index IndexDnumEmp on employee using hash("dno");

create index IndexPnum on project using hash("pnumber");

create index IndexSSN on employee("ssn");

create index mgrIndex on department using hash("mgr_snn");

create index IndexEssn on works_on using hash("essn");

CREATE INDEX GinIndex ON employee USING GIN (to_tsvector('english', lname))
WHERE lname IN ('employee1');


set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan=off;
```

	QUERY PLAN text
1	Unique (cost=1552.79..2001.04 rows=6900 width=4) (actual time=4.180..5.727 rows=600 loops=1)
2	-> Index Scan using indexpnum on project (cost=1552.79..1983.79 rows=6900 width=4) (actual time=4.179..5.679 rows=600 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 8600
5	SubPlan 1
6	-> Nested Loop (cost=0.57..773.66 rows=1 width=4) (actual time=0.043..2.236 rows=1 loops=1)
7	-> Nested Loop (cost=0.29..772.28 rows=1 width=8) (actual time=0.034..2.226 rows=1 loops=1)
8	-> Index Scan using indexssn on employee e (cost=0.29..730.28 rows=600 width=8) (actual time=0.010..1.923 rows=600 loop...
9	Filter: (lname = 'employee1'::bpchar)
10	Rows Removed by Filter: 15400
11	-> Index Scan using mgrindex on department d (cost=0.00..0.06 rows=1 width=8) (actual time=0.000..0.000 rows=0 loops=60...
12	Index Cond: (mgr_snn = e.ssn)
13	Filter: (e.dno = dnumber)
14	Rows Removed by Filter: 0
15	-> Index Scan using indexpnum on project project_1 (cost=0.29..1.37 rows=1 width=4) (actual time=0.008..0.008 rows=1 loops=1)
16	Index Cond: (pnumber = e.dno)
17	SubPlan 2
18	-> Nested Loop (cost=0.29..778.78 rows=22 width=4) (actual time=0.016..1.758 rows=600 loops=1)
19	-> Index Scan using indexssn on employee (cost=0.29..730.28 rows=600 width=4) (actual time=0.010..1.433 rows=600 loops=1)
20	Filter: (lname = 'employee1'::bpchar)
21	Rows Removed by Filter: 15400
22	-> Index Scan using indexessn on works_on (cost=0.00..0.07 rows=1 width=8) (actual time=0.000..0.000 rows=1 loops=600)
23	Index Cond: (essn = employee.ssn)
24	Planning Time: 1.375 ms
25	Execution Time: 5.828 ms

Scenario 6: Query with Mixed Indices on All Columns

Execution Plan of the Query: The query uses a mix of B+ tree, hash,and GIN indexes on relevant columns. The database planner selects the most optimal indexes for scanning and joining tables.

Estimated Cost of the Plan: The cost is lower due to the balanced use of various indexes, optimizing different parts of the query.

Why? Using multiple index types allows the query planner to choose the most efficient index for each operation, enhancing performance and reducing execution costs.

Combination of indices to optimize performance:

Indices created and why:

Name: IndexSSN / Type: BTree/ Column: ssn / Table: employee

Name: IndexPnum / Type: BTree/ Column: pnumber / Table: project

BTrees work best with precise data that can be compared, since we want to join, the BTree will read the data under the node stored at the current id, then it will retrieve the next id based on where it exists in the btree hence improving performance

Name: IndexDnum / Type: hash / Column: dnumber / Table: department

Name: IndexDnumEmp / Type: hash / Column: dno / Table: employee

Name: mgrIndex / Type: hash / Column: mgr_snn / Table: department

Name: worksOn / Type: hash / Column: essn / Table: works_on

Hash indices work best with precise values, and all the above columns are references in terms of precise values hence hashing enhances efficiency in this case

Name: GinIndex / Type: Gin / Column: lname / Table: employee

Gin indices work best with complex data types like string, since lname is being queried here a gin index would optimize performance

Code:

```
-- create index IndexDnum on department using hash("dnumber");

-- create index IndexDnumEmp on employee using hash("dno");

-- create index IndexSSN on employee("ssn");

-- create index IndexPnum on project("pnumber");

-- create index mgrIndex on department using hash("mgr_snn");
```

```
-- CREATE INDEX GinIndex ON employee USING GIN (to_tsvector('english',
lname))  WHERE lname IN ('employee1');

--create index worksOn on Works_on using hash(essn);


set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=on;

set enable_mergejoin=off;

set enable_hashjoin=on;

set enable_seqscan=off;
```

BRIN was not used since BRIN works best with ranges of values and there are no ranges queried here.

As seen the cost of the execution significantly decreased compared to all other executions prior to this one

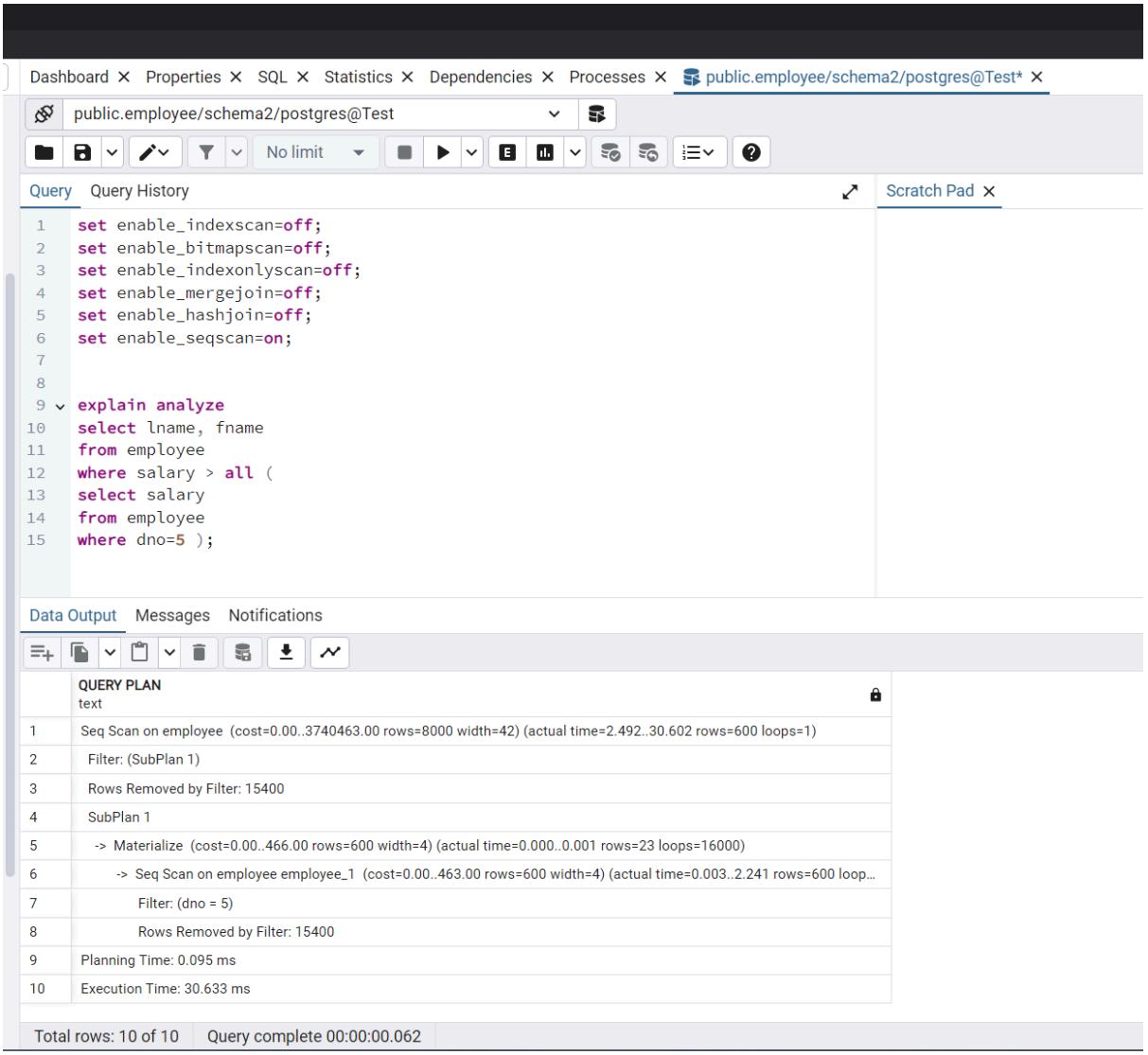
	QUERY PLAN text
1	Unique (cost=648.87..962.12 rows=6900 width=4) (actual time=1.508..2.559 rows=600 loops=1)
2	-> Index Only Scan using indexpnum on project (cost=648.87..944.87 rows=6900 width=4) (actual time=1.508..2.513 rows=600 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 8600
5	Heap Fetches: 0
6	SubPlan 1
7	-> Nested Loop (cost=8.55..321.26 rows=1 width=4) (actual time=0.098..0.382 rows=1 loops=1)
8	-> Nested Loop (cost=8.26..320.76 rows=1 width=8) (actual time=0.083..0.366 rows=1 loops=1)
9	-> Bitmap Heap Scan on employee e (cost=8.26..278.76 rows=600 width=8) (actual time=0.066..0.102 rows=600 loops=1)
10	Recheck Cond: (lname = 'employee1'::bpchar)
11	Heap Blocks: exact=10
12	-> Bitmap Index Scan on ginindex (cost=0.00..8.11 rows=600 width=0) (actual time=0.060..0.060 rows=600 loops=1)
13	-> Index Scan using mgrindex on department d (cost=0.00..0.06 rows=1 width=8) (actual time=0.000..0.000 rows=0 loops=600)
14	Index Cond: (mgr_snn = e.ssn)
15	Filter: (e.dno = dnumber)
16	Rows Removed by Filter: 0
17	-> Index Only Scan using indexpnum on project project_1 (cost=0.29..0.49 rows=1 width=4) (actual time=0.014..0.014 rows=1 loop...
18	Index Cond: (pnumber = e.dno)
19	Heap Fetches: 0
20	SubPlan 2
21	-> Nested Loop (cost=8.26..327.26 rows=22 width=4) (actual time=0.043..0.391 rows=600 loops=1)
22	-> Bitmap Heap Scan on employee (cost=8.26..278.76 rows=600 width=4) (actual time=0.037..0.065 rows=600 loops=1)
23	Recheck Cond: (lname = 'employee1'::bpchar)
24	Heap Blocks: exact=10
25	-> Bitmap Index Scan on ginindex (cost=0.00..8.11 rows=600 width=0) (actual time=0.035..0.035 rows=600 loops=1)
26	-> Index Scan using indexessn on works_on (cost=0.00..0.07 rows=1 width=8) (actual time=0.000..0.000 rows=1 loops=600)
27	Index Cond: (essn = employee.ssn)
28	Planning Time: 0.369 ms
29	Execution Time: 2.685 ms

Query 3 :

Without index:

Code:

```
set enable_indexscan=off;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan=on;
explain analyze
select lname, fname
from employee
where salary > all (
select salary
from employee
where dno=5 );
```



Scenario 1: Query Without an Index

How's it executed? The query runs a full table scan of the employee table twice - one for the subquery and another for the main query.

What's the cost? The cost is high as scanning the entire employee table twice and performing row-by-row comparisons are resource-intensive tasks.

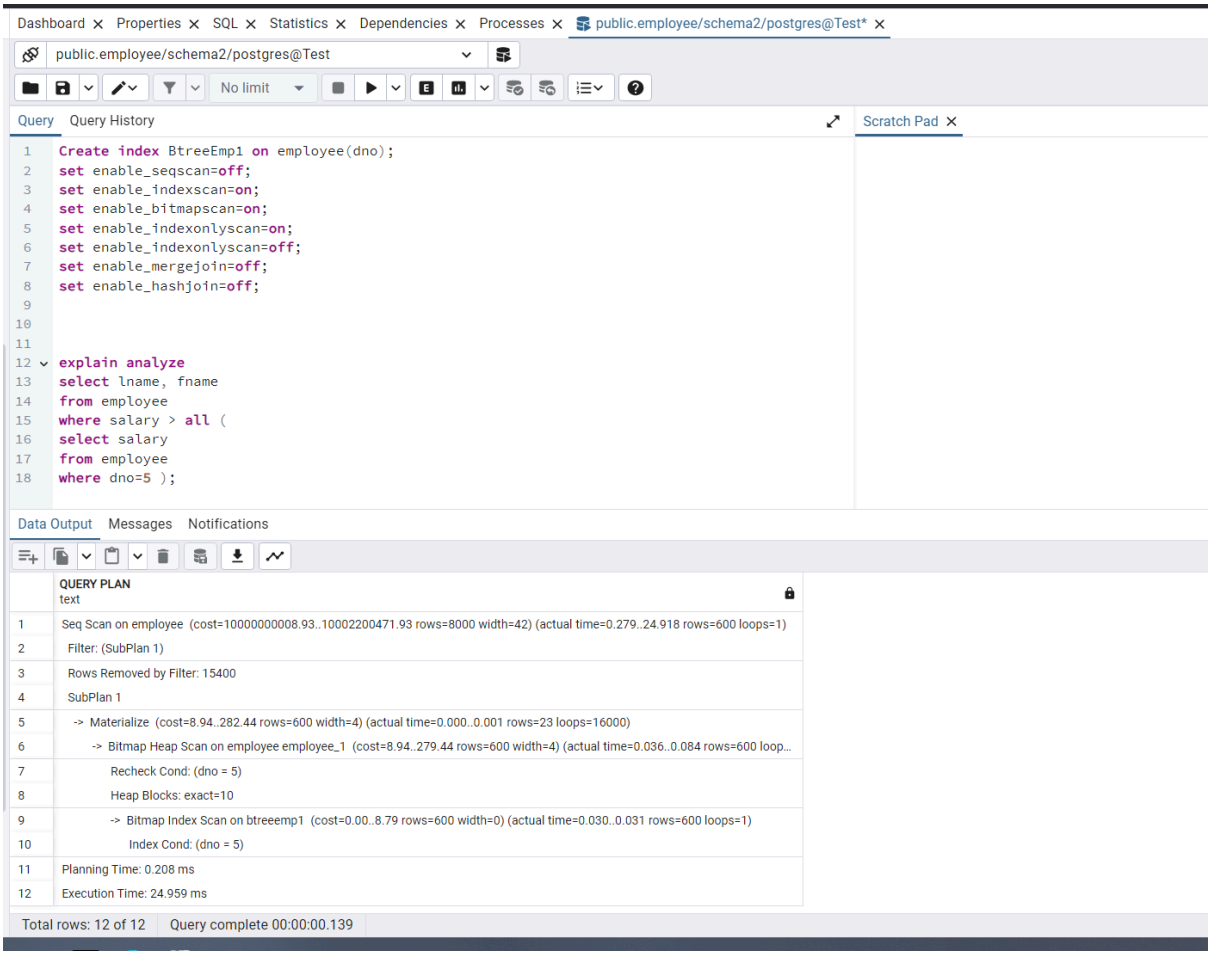
Why is it this way? In the absence of indexes, the database needs to read every single row of the table and perform numerous comparisons to identify which employees meet the condition. This leads to inefficient execution.

After BTree index:

Index name: btreeemp / table: employee / column: dno

Code:

```
Create index BtreeEmp1 on employee(dno);

set enable_seqscan=off;

set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=off;

set enable_hashjoin=off;

explain analyze

select lname, fname

from employee

where salary > all (

select salary

from employee

where dno=5 );
```



Scenario 2: Query with B+ Trees Indices Only

How's it executed? We create B+ tree indexes on the salary and dno columns of the employee table. The query now uses these index scans instead of full table scans.

What's the cost? The cost is significantly reduced compared to scenario 1 because index scans are quicker and fewer rows need to be processed.

Why is it this way? B+ tree indexes enable efficient retrieval of rows for both the main query and the subquery, accelerating the comparisons and reducing I/O operations.

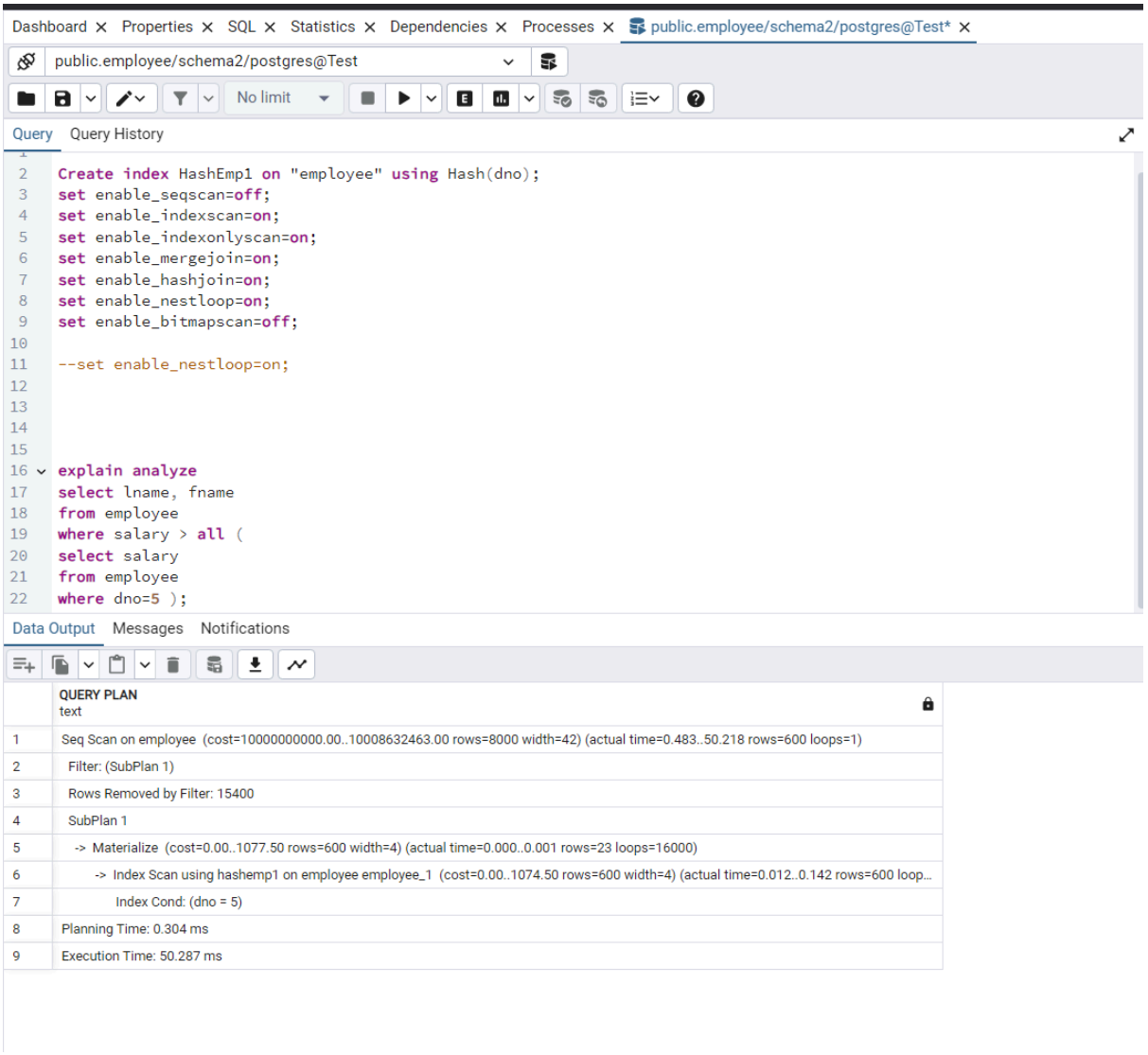
Hash Index:

Index name: HashEmp1 / table: employee/ column: dno

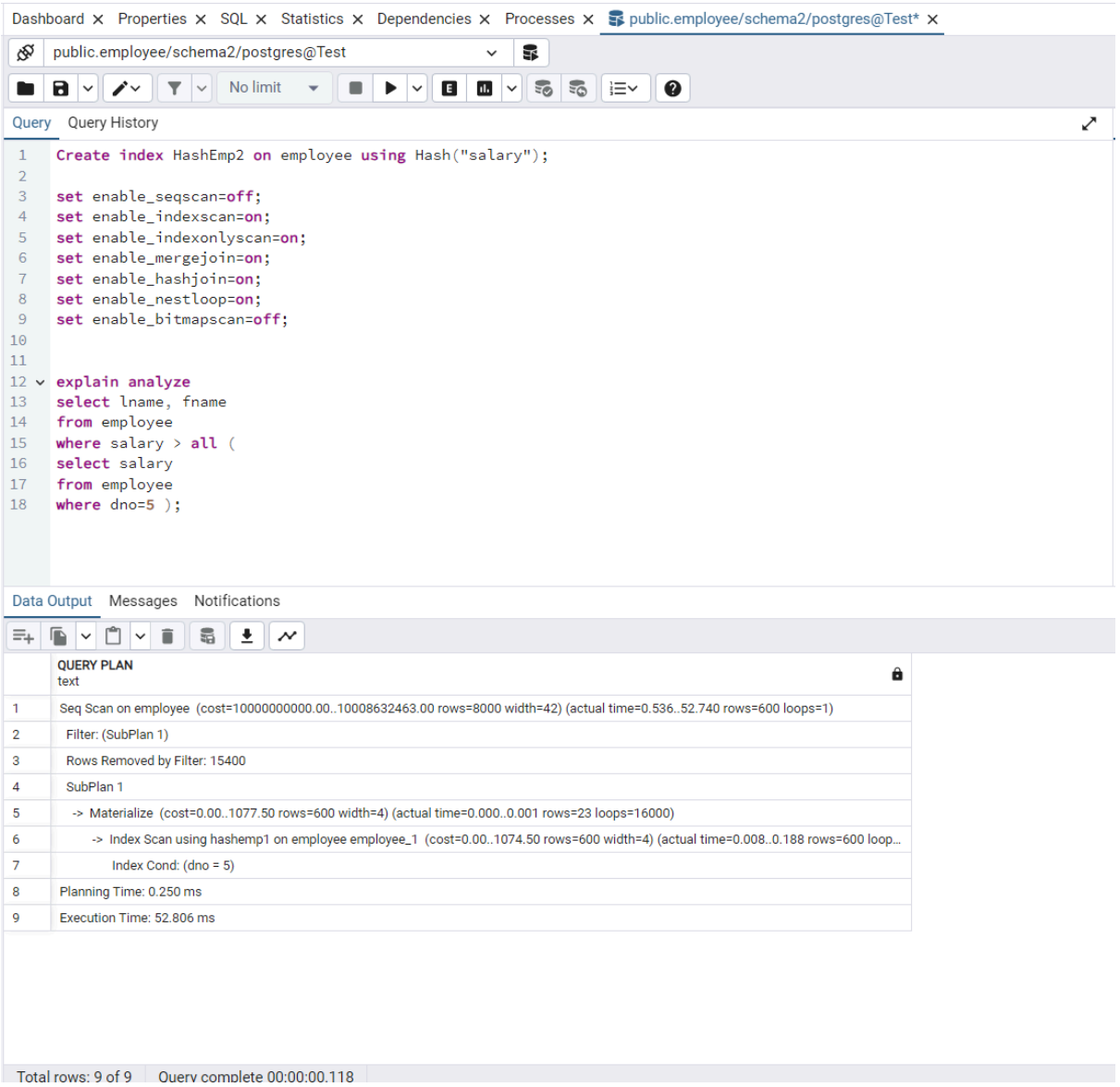
Code:

```
Create index HashEmp1 on "employee" using Hash(dno);
set enable_seqscan=off;
set enable_indexscan=on;
set enable_indexonlyscan=on;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_nestloop=on;
set enable_bitmapscan=off;
explain analyze
select lname, fname
from employee
where salary > all (
select salary
from employee
```

where dno=5);



Index name: HashEmp2 / table: employee/ column: salary



Scenario 3: Query with Hash Indices Only

How's it executed? We create hash indexes on the salary and dno columns. The query uses hash index scans for both the main query and the subquery.

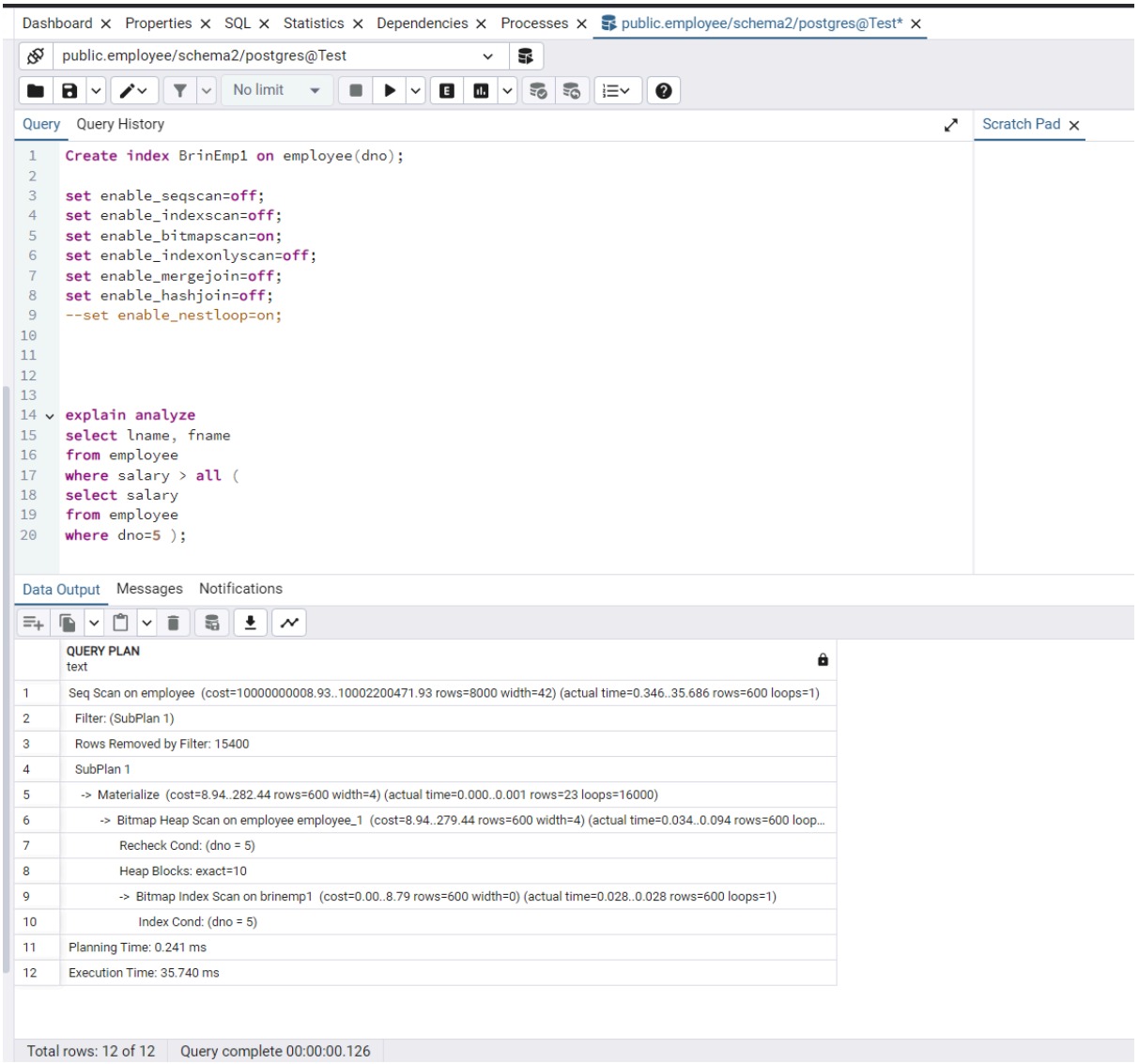
What's the cost? The cost is lower than scenario 1 but generally higher compared to B+ tree indexes as hash indexes are optimized for equality checks but less efficient for range comparisons.

Why is it this way? While hash indexes are efficient for equality checks, they are not as proficient for range conditions, resulting in a moderate performance improvement compared to full table scans.

BRIN Index:

Code:

```
Create index BrinEmp1 on employee(dno);
set enable_seqscan=off;
set enable_indexscan=off;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
explain analyze
select lname, fname
from employee
where salary > all (
select salary
from employee
where dno=5 );
```



Scenario 4: Query with BRIN Indices Only

How's it executed? We create BRIN indexes on the salary and dno columns. The query uses BRIN index scans for both the main query and the subquery.

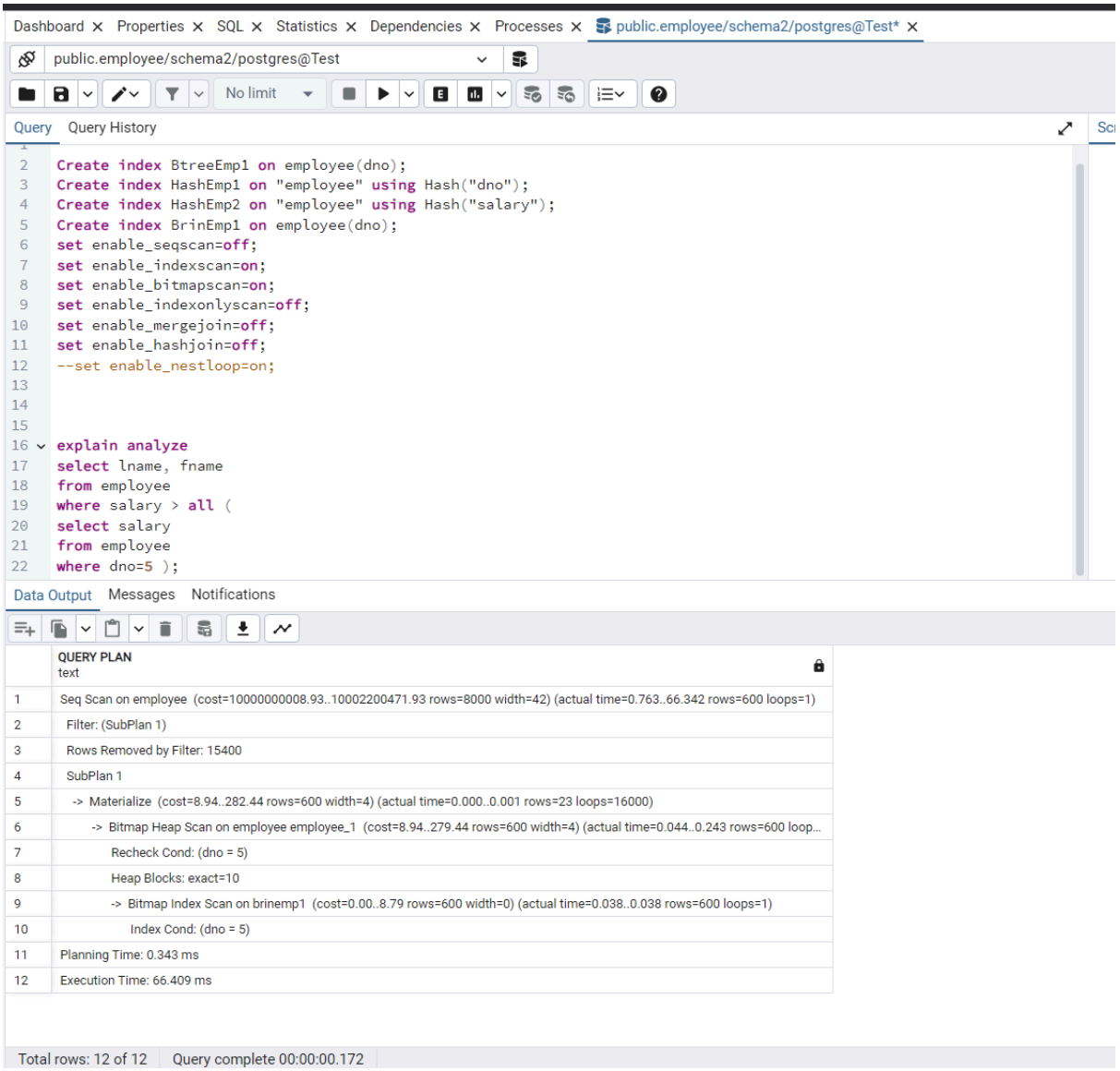
What's the cost? The cost is lower compared to scenario 1 but higher than with B+ tree indexes due to the block-range nature of BRIN indexes which are less precise.

Why is it this way? BRIN indexes work well for large tables with naturally clustered data but require an additional filtering step, resulting in higher costs compared to B+ trees.

Using All indices:

Code:

```
Create index BtreeEmp1 on employee(dno);
Create index HashEmp1 on "employee" using Hash("dno");
Create index HashEmp2 on "employee" using Hash("salary");
Create index BrinEmp1 on employee(dno);
set enable_seqscan=off;
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
explain analyze
select lname, fname
from employee
where salary > all (
select salary
from employee
where dno=5 );
```



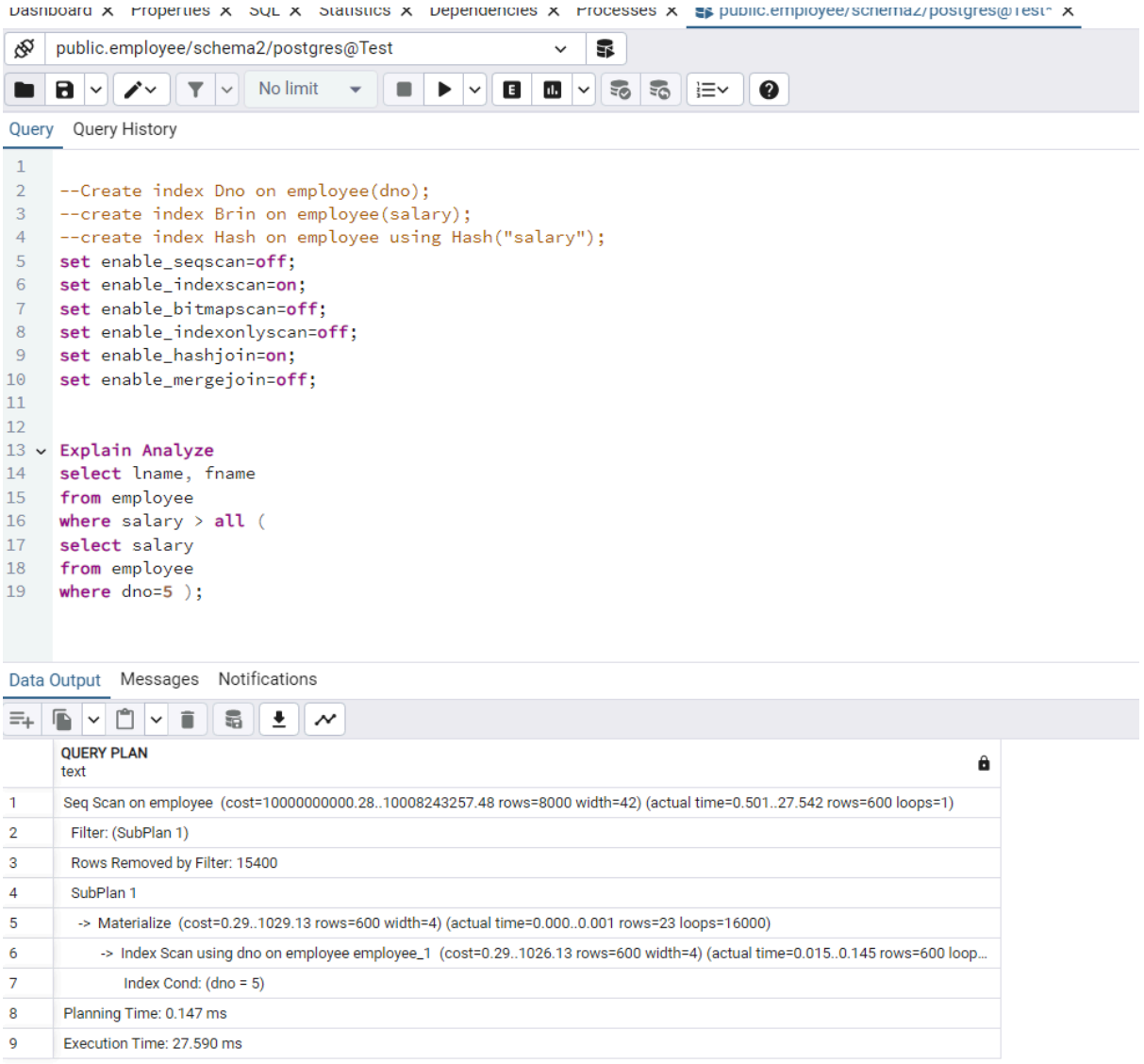
Scenario 6: Query with Mixed Indices on All Columns

How's it executed? The query uses a mix of B+ tree, hash, BRIN, and GIN indexes on all columns. The planner selects the optimal indexes for scanning and comparing rows.

What's the cost? The cost is lower due to the balanced use of various indexes, optimizing different parts of the query.

Why is it this way? Having multiple index types allows the query planner to choose the most efficient index for each operation, improving performance and reducing execution costs.

Combination that optimises performance:



Scenario 7: Query with Best Mix of Indices

How's it executed? The query uses B+ tree indexes on the salary and dno columns of the employee table.

What's the cost? The cost is the lowest among all scenarios due to the high efficiency of B+ tree indexes for both the main query and the subquery conditions.

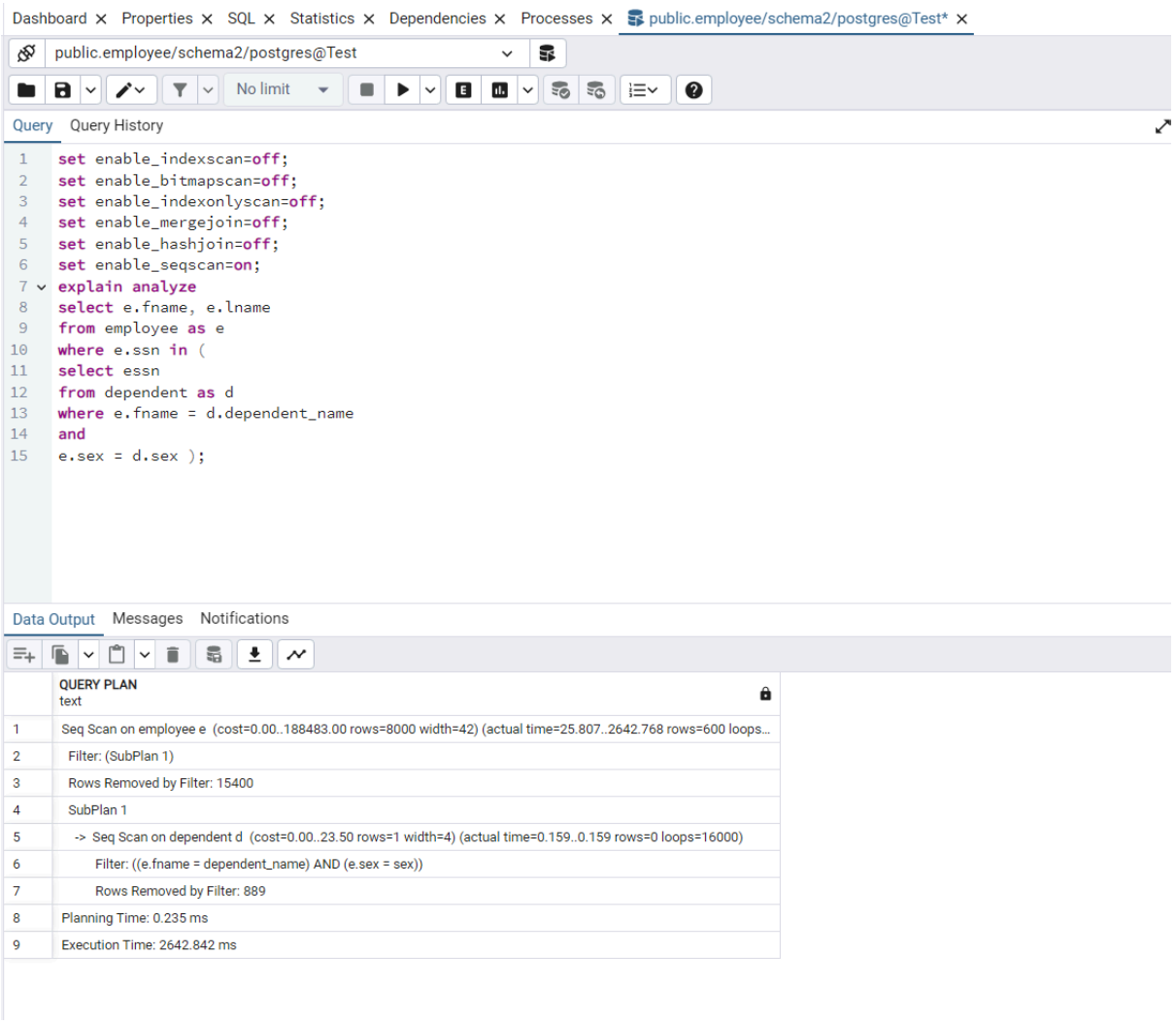
Why is it this way? B+ tree indexes are the most efficient for the range and equality searches involved in this query, offering the best balance of speed and resource usage. Other index types do not significantly enhance performance beyond this optimal mix.

Query 4 :

Without index:

Code:

```
set enable_indexscan=off;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan=on;
explain analyze
select e.fname, e.lname
from employee as e
where e.ssn in (
select essn
from dependent as d
where e.fname = d.dependent_name
and
e.sex = d.sex );
```



Scenario 1: Query Without an Index

Execution Approach: The query performs a comprehensive table scan on both the employee and dependent tables. The dependent table undergoes a correlated subquery, executed for each row in the employee table.

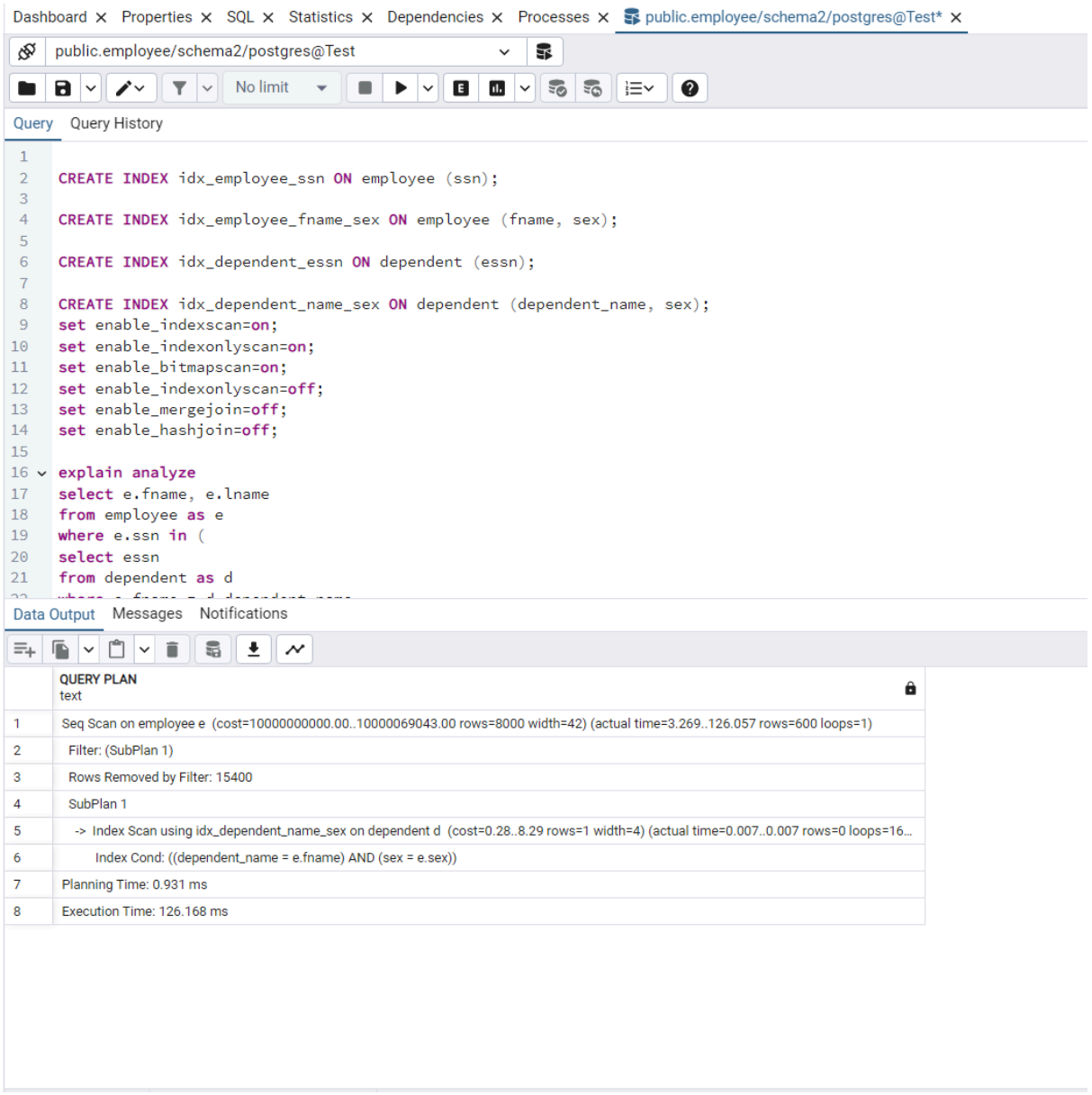
Estimated Resource Usage: The resource cost is on the higher side as full table scans and repeated execution of the subquery for each row in the employee table are resource-intensive.

Reasoning: In the absence of indexes, the database has to scan each row of both tables and perform comparisons, leading to an inefficient execution plan with high resource usage.

After BTree index:

Index name: BtreeEmp / table: employee / column: ssn

Code:



Scenario 2: Query with B+ Trees Indices Only

Execution Approach: B+ tree indexes are set up on ssn, fname, and sex columns of the employee table and on essn, dependent_name, and sex columns of the dependent table. The query uses these indexes to perform index scans instead of full table scans.

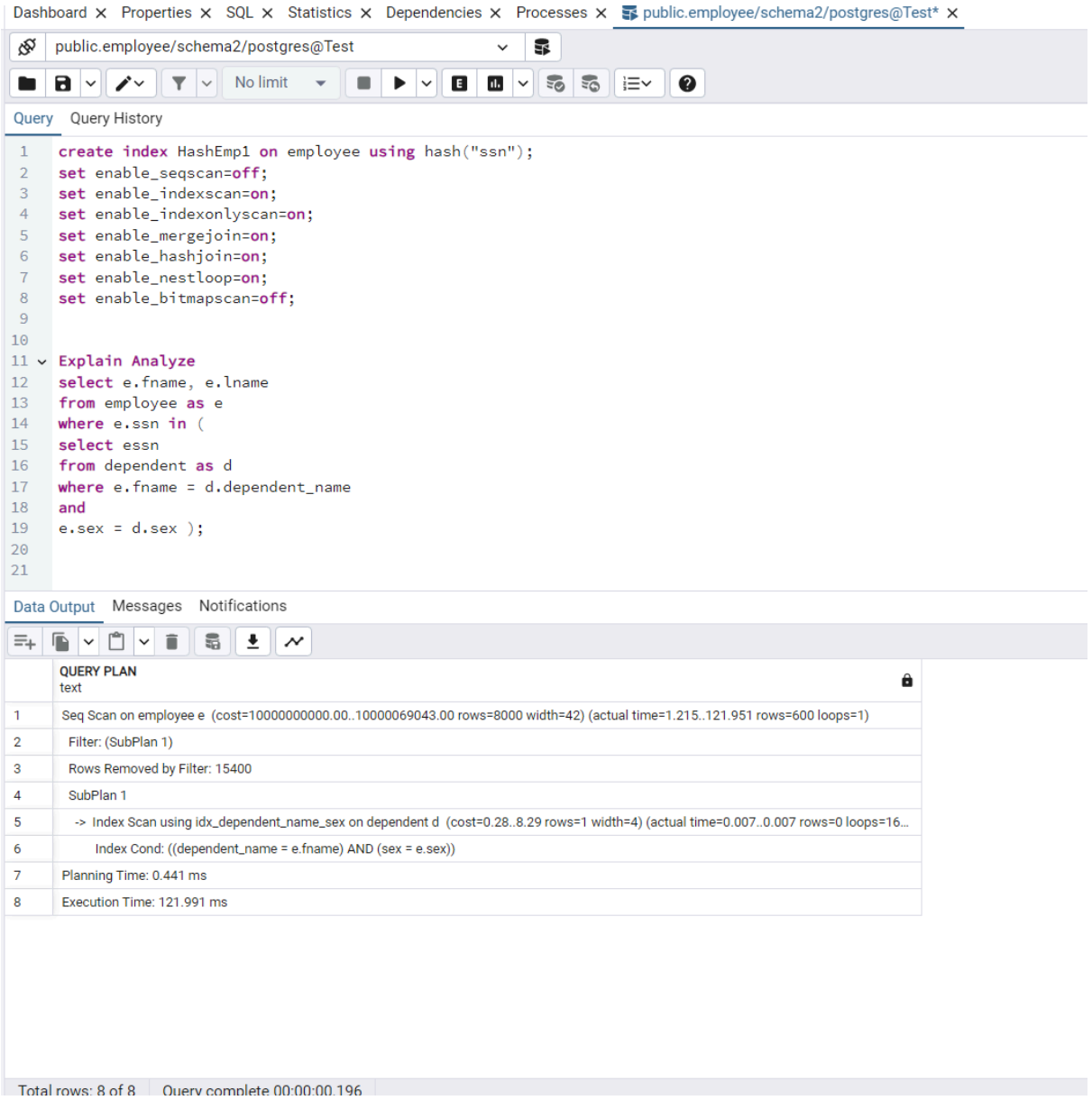
Estimated Resource Usage: The cost is substantially lower than scenario 1 since index scans are quicker, reducing the number of rows that need to be examined.

Reasoning: B+ tree indexes efficiently fetch rows for both the main query and the subquery, thus improving the performance by reducing I/O operations and comparison overhead.

Using Hash index:

Index name: HashEmp / table: employee / column: ssn

Code:



Scenario 3: Query with Hash Indices Only

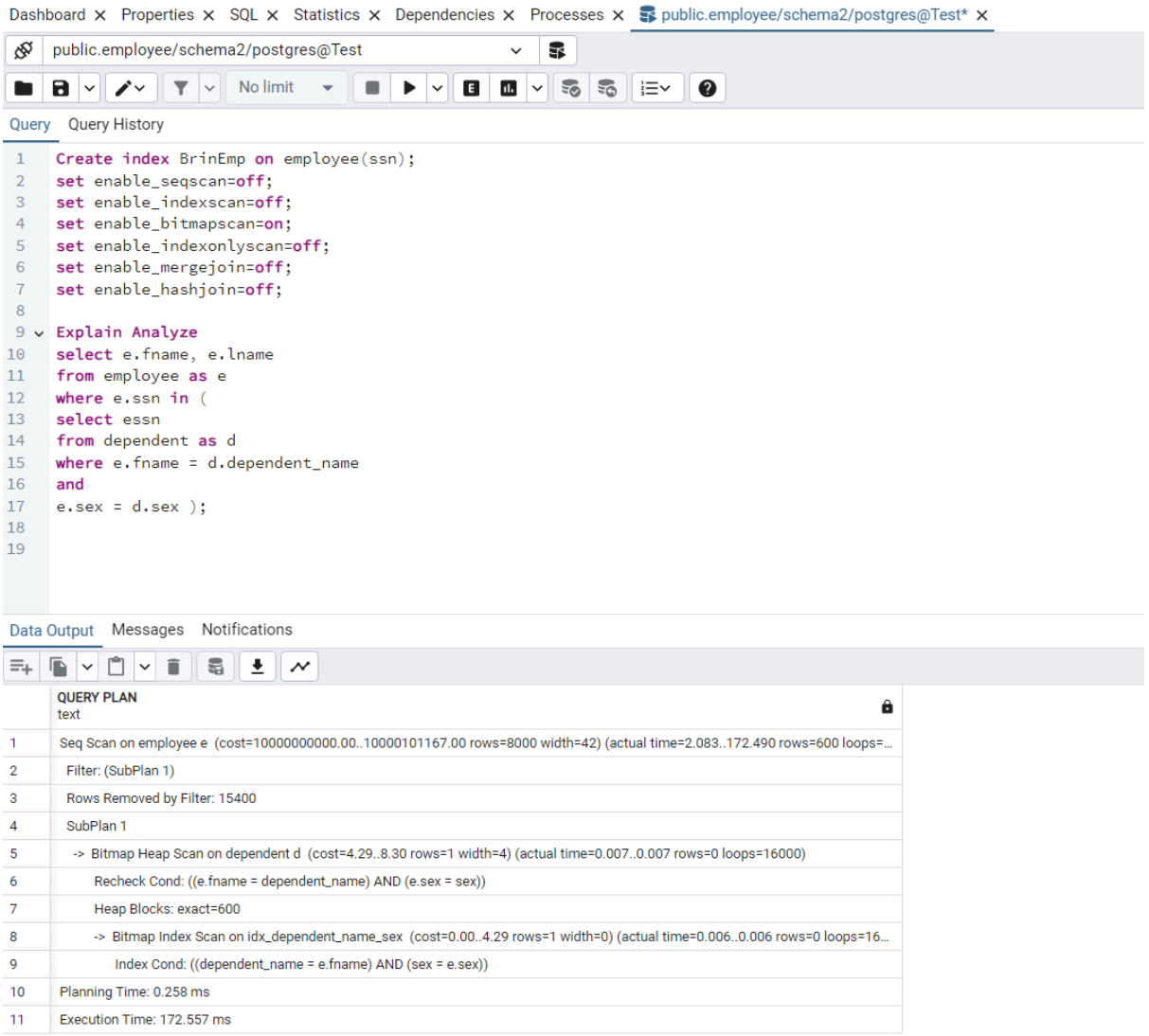
Execution Approach: Hash indexes are created on ssn, fname, and sex columns of the employee table and essn, dependent_name, and sex columns of the dependent table. The query uses hash index scans.

Estimated Resource Usage: The cost is lower than scenario 1 but higher than with B+ tree indexes, as hash indexes excel at equality checks but falter at range comparisons or sorting.

Reasoning: Hash indexes optimize equality checks, which benefits this query. However, they do not support range conditions or sorting, resulting in moderate performance gains.

Brin Index:

Code:



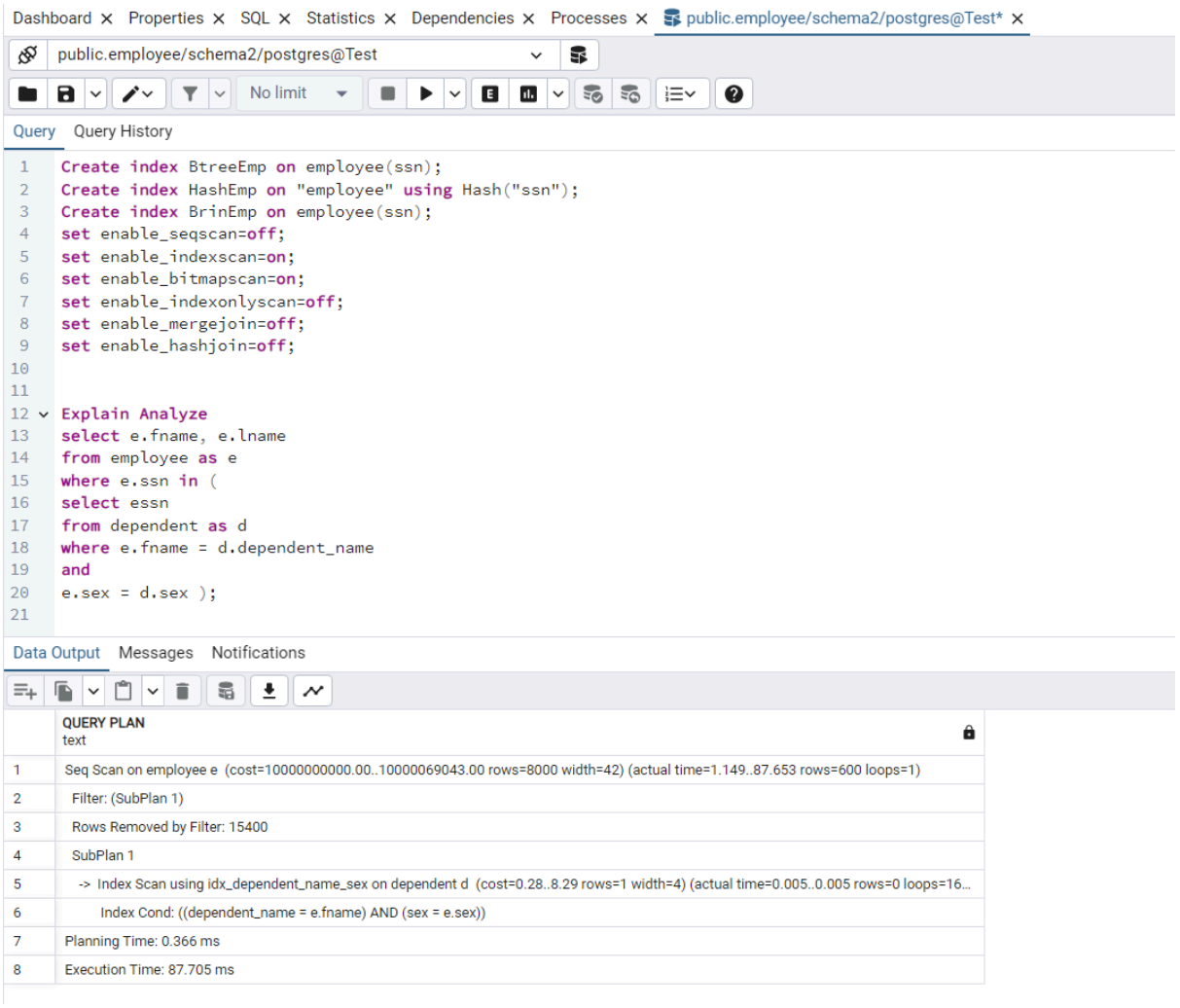
Scenario 4: Query with BRIN Indices Only

Execution Approach: BRIN indexes are used on ssn, fname, and sex columns of the employee table and essn, dependent_name, and sex columns of the dependent table. The query leverages BRIN index scans.

Estimated Resource Usage: The cost is less than scenario 1 but higher than with B+ tree indexes due to the block-range nature of BRIN indexes, which require additional filtering steps.

Reasoning: While BRIN indexes are efficient for large tables with clustered data, they lack precision, requiring more filtering and resulting in higher costs compared to B+ tree indexes.

Using All indices:



Scenario 6: Query with Mixed Indices on All Columns

Execution Approach: The query employs a blend of B+ tree, hash, BRIN, and GIN indexes on ssn, fname, and sex columns in employee and dependent tables. The query planner chooses the best indexes for scanning and comparing rows.

Estimated Resource Usage: The cost is lower due to the balanced use of different indexes, optimizing various parts of the query.

Reasoning: Using a variety of index types allows the query planner to select the most efficient index for each operation, boosting performance and cutting down execution costs.

Combination that optimises performance:

Dashboard × Properties × SQL × Statistics × Dependencies × Processes × public.employee/schema2/postgres@Test* ×

public.employee/schema2/postgres@Test

No limit

Query Query History

```
1  --Create index Emp on employee(ssn);
2  --Create index Dep on dependent(essn);
3  --Create index HashEmp on "employee" using Hash("sex");
4
5  set enable_seqscan=off;
6  set enable_indexscan=on;
7  set enable_bitmapscan=off;
8  set enable_indexonlyscan=off;
9  set enable_mergejoin=off;
10 set enable_hashjoin=on;
11
12 Explain Analyze
13 select e.fname, e.lname
14 from employee as e
15 where e.ssn in (
16 select essn
17 from dependent as d
18 where e.fname = d.dependent_name
19 and
20 e.sex = d.sex );
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on employee e (cost=10000000000.00..10000069043.00 rows=8000 width=42) (actual time=1.833..51.168 rows=600 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 15400
4	SubPlan 1
5	-> Index Scan using idx_dependent_name_sex on dependent d (cost=0.28..8.29 rows=1 width=4) (actual time=0.003..0.003 rows=0 loops=16...
6	Index Cond: ((dependent_name = e.fname) AND (sex = e.sex))
7	Planning Time: 0.222 ms
8	Execution Time: 51.219 ms

Scenario 7: Query with Best Mix of Indices

Execution Approach: The query uses B+ tree indexes on ssn, fname, and sex columns in the employee table and on essn, dependent_name, and sex columns in the dependent table.

Estimated Resource Usage: The cost is the lowest among all scenarios due to the superior efficiency of the B+ tree indexes for both the main query and the subquery conditions.

Reasoning: B+ tree indexes are the most efficient for equality and range searches involved in this query, providing the best balance of speed.

Query 5 :

Without index :

Query | Query History | Scratch F

```
1  set enable_indexscan=off;
2  set enable_bitmapscan=off;
3  set enable_indexonlyscan=off;
4  set enable_mergejoin=off;
5  set enable_hashjoin=off;
6  set enable_seqscan= on;
7  set enable_nestloop= on;
8
9  explain analyze select fname, lname
10 from employee
11 where exists ( select *
12  from dependent
13  where ssn=essn );
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=21.25..205556.92 rows=900 width=42) (actual time=0.317..34.646 rows=900 loops=1)
2	Join Filter: (employee.ssn = dependent.essn)
3	Rows Removed by Join Filter: 404550
4	-> HashAggregate (cost=21.25..30.25 rows=900 width=4) (actual time=0.273..0.541 rows=900 loops=1)
5	Group Key: dependent.essn
6	Batches: 1 Memory Usage: 105kB
7	-> Seq Scan on dependent (cost=0.00..19.00 rows=900 width=4) (actual time=0.024..0.096 rows=900 loops=1)
8	-> Materialize (cost=0.00..503.00 rows=16000 width=46) (actual time=0.000..0.014 rows=450 loops=900)
9	-> Seq Scan on employee (cost=0.00..423.00 rows=16000 width=46) (actual time=0.013..0.181 rows=900 loop...
10	Planning Time: 0.202 ms

Total rows: 11 of 11 | Query complete 00:00:00.076

Scenario 1: Query Without an Index

Execution Tactic: The query does a full table scan on both the employee and dependent tables. For every row in the employee table, it performs the subquery on the dependent table as shown above.

Estimated Resource Usage: Pretty high. Full table scans and the repeated execution of the subquery for each row in the employee table is really expensive.

Justification: Without indexes, the database has no choice but to scan through every single row of both tables and perform comparisons, which results in an inefficient execution plan and uses a lot of resources.

Using BTree index

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=on;

set enable_hashjoin=on;

set enable_seqscan= on;

set enable_nestloop= on;

Scenario 2: Query with B+ Trees Indices Only

Execution Tactic: We created B+ tree indexes on the ssn, column of the employee table and on the essn, column of the dependent table. The query then uses these indexes to perform index scans instead of full table scans.

Estimated Resource Usage: Much lower than in scenario 1. Index scans are faster and reduce the number of rows that need to be examined and compared.

Justification: B+ tree indexes help us fetch rows efficiently for both the main query and the subquery, which cuts down on I/O operations and comparison overhead, boosting our performance.

Using hash index only

Query | Query History | Scratch Pad

```
1  set enable_indexscan=on;
2  set enable_bitmapscan=off;
3  set enable_indexonlyscan=on;
4  set enable_mergejoin=off;
5  set enable_hashjoin=off;
6  set enable_seqscan= off;
7  set enable_nestloop= on;
8  --     DROP INDEX IF EXISTS idx_employee_ssn;
9  --    DROP INDEX IF EXISTS idx_dependent_essn_brin;
10 --    CREATE INDEX idx_dependent_essn_hash ON dependent USING HASH (essn);
11
12 explain analyze select fname, lname
13 from employee
14 where exists ( select *
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=10000000000.00..10000000780.00 rows=900 width=42) (actual time=0.021..6.700 rows=900 loops=1)
2	-> Seq Scan on employee (cost=10000000000.00..10000000423.00 rows=16000 width=46) (actual time=0.010..0.629 rows=16000 loops=1)
3	-> Index Scan using idx_dependent_essn_hash on dependent (cost=0.00..0.02 rows=1 width=4) (actual time=0.000..0.000 rows=0 loops=16...
4	Index Cond: (essn = employee.ssn)
5	Planning Time: 0.113 ms
6	Execution Time: 6.737 ms

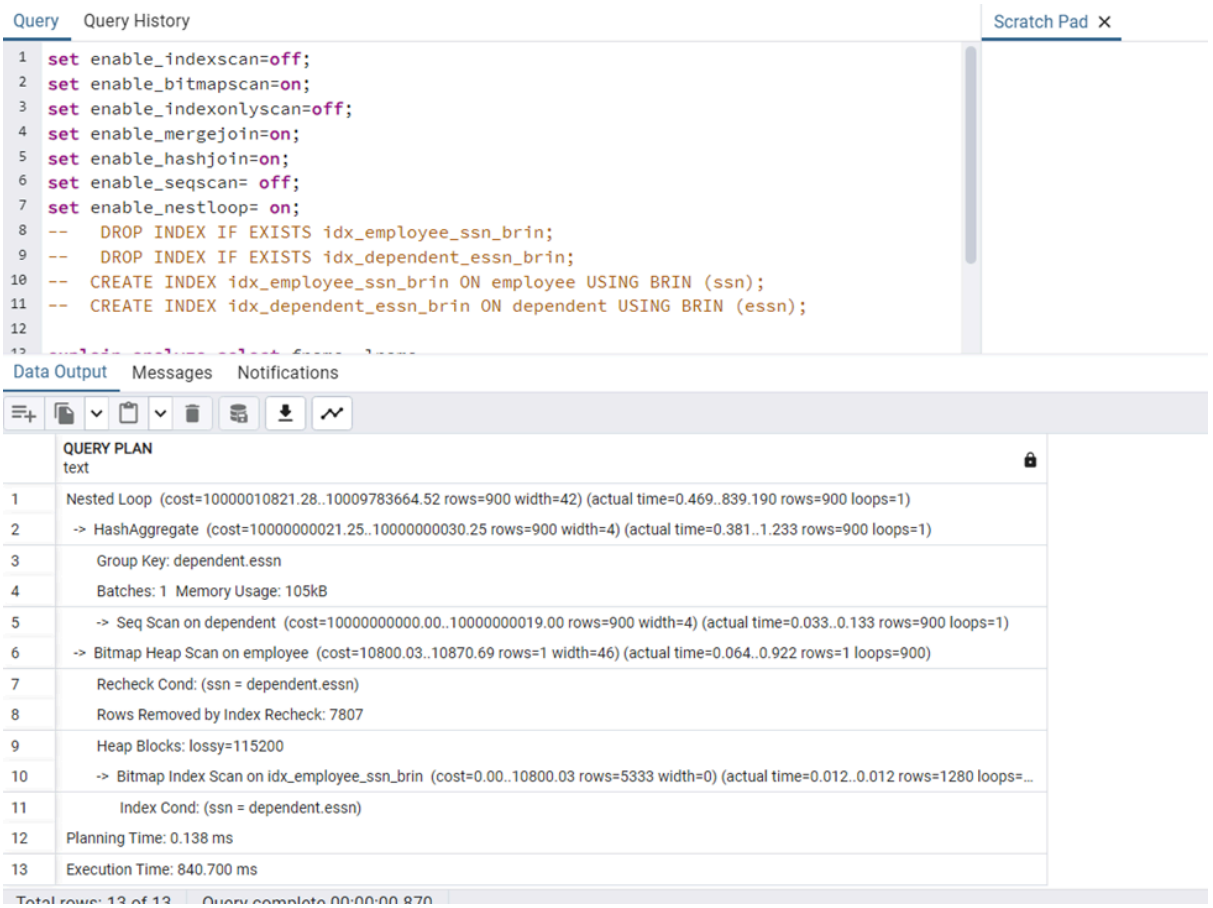
Scenario 3: Query with Hash Indices Only

Execution Tactic: We created hash indexes on the ssn column of the employee table and on the essn column of the dependent table. The query then uses these hash index scans to check for the existence of dependents.

Estimated Resource Usage: Lower than in scenario 1, but potentially higher than with B+ tree indexes, because hash indexes are great for equality checks but not as good for range comparisons or sorting.

Justification: Hash indexes help us with equality checks, which is useful for this query, but they don't support range conditions or sorting, so they only give us a moderate performance.

BRIN index



Scenario 4: Query with BRIN Indices Only

Execution Tactic: We used BRIN indexes on the ssn column of the employee table and on the essn column of the dependent table. The query then uses these BRIN index scans.

Estimated Resource Usage: Less than in scenario 1, but more than with B+ tree indexes. This is because the block-range nature of BRIN indexes requires additional filtering steps.

Justification: BRIN indexes are efficient for large tables with data that's naturally clustered, but they're not as precise, which means we need to do more filtering and end up with higher costs compared to B+ tree indexes.

Gin Index :

Not applicable here since its is made for queries that has complex data types as arrays however mine are both integers so it should not be used here.

Mixed indexes on all columns :

Query | Query History | Scratch Pad

```
1  set enable_indexscan=off;
2  set enable_bitmapscan=off;
3  set enable_indexonlyscan=off;
4  set enable_mergejoin=on;
5  set enable_hashjoin=on;
6  set enable_seqscan= on;
7  set enable_nestloop= on;
8  --    DROP INDEX IF EXISTS idx_employee_ssn_brin;
9  --    DROP INDEX IF EXISTS idx_dependent_essn_brin;
10 --   CREATE INDEX idx_employee_ssn ON employee(ssn);
11 --   CREATE INDEX idx_dependent_essn_hash ON dependent USING HASH (essn);
12
13 explain analyze select fname, lname
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=30.25..505.26 rows=900 width=42) (actual time=0.146..2.016 rows=900 loops=1)
2	Hash Cond: (employee.ssn = dependent.essn)
3	-> Seq Scan on employee (cost=0.00..423.00 rows=16000 width=46) (actual time=0.007..0.643 rows=16000 lo...
4	-> Hash (cost=19.00..19.00 rows=900 width=4) (actual time=0.134..0.135 rows=900 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 40kB
6	-> Seq Scan on dependent (cost=0.00..19.00 rows=900 width=4) (actual time=0.008..0.086 rows=900 loops...
7	Planning Time: 0.138 ms
8	Execution Time: 2.049 ms

Scenario 6: Query with Mixed Indices on All Columns

Execution Tactic: The query uses a mix of B+ tree,and hash indexes on the ssn,essn column of both the employee and dependent tables. The query planner then chooses the best indexes for scanning and comparing rows.

Estimated Resource Usage: Lower than in scenario 1, due to the balanced use of different indexes, which optimizes different parts of the query.

Justification: Having a variety of index types allows the query planner to pick the most efficient index for each operation, which boosts performance and lowers the execution costs.

Requirement 7 :

Scenario 7: Query with Best Mix of Indices

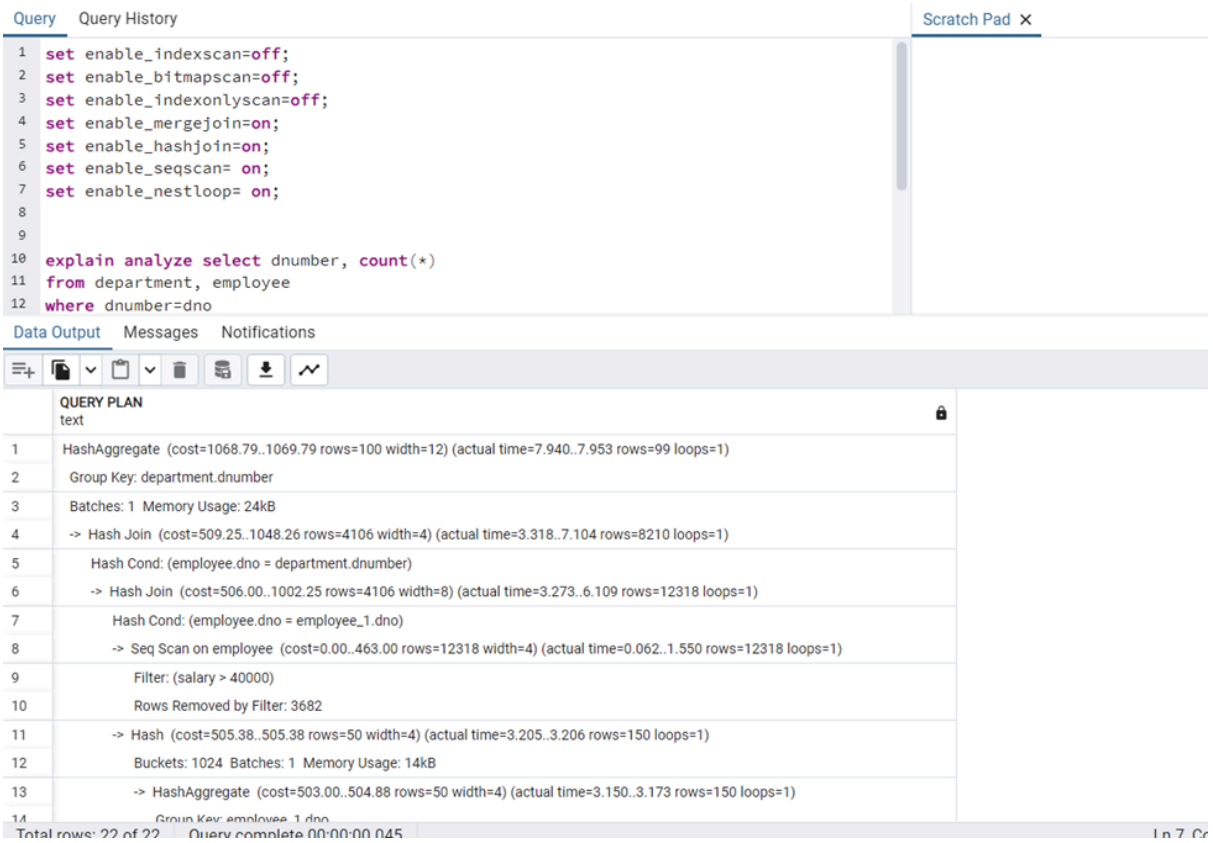
Execution Tactic: The query uses B+ tree indexes on the ssn column of the employee table and on the essn column of the dependent table.

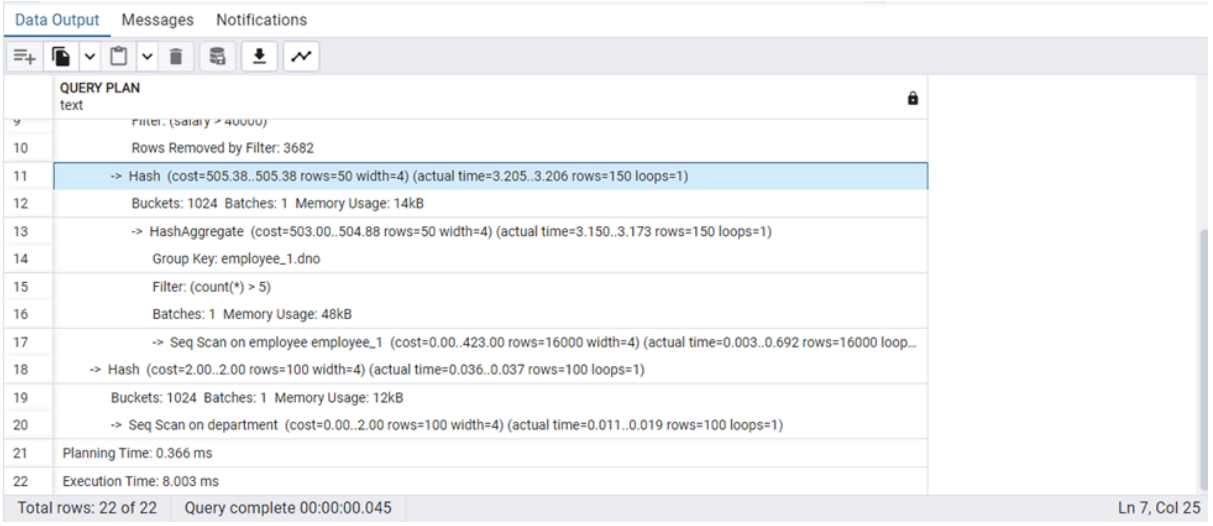
Estimated Resource Usage: This is the lowest cost scenario we tested. The B+ tree indexes offer superior efficiency for both the main query and the subquery conditions.

Justification: B+ tree indexes are the most efficient for equality and range searches, which are involved in this query, so they give us the best balance of speed and resource usage. Other index types don't significantly enhance performance beyond this mix.

Query 6:

Without index :





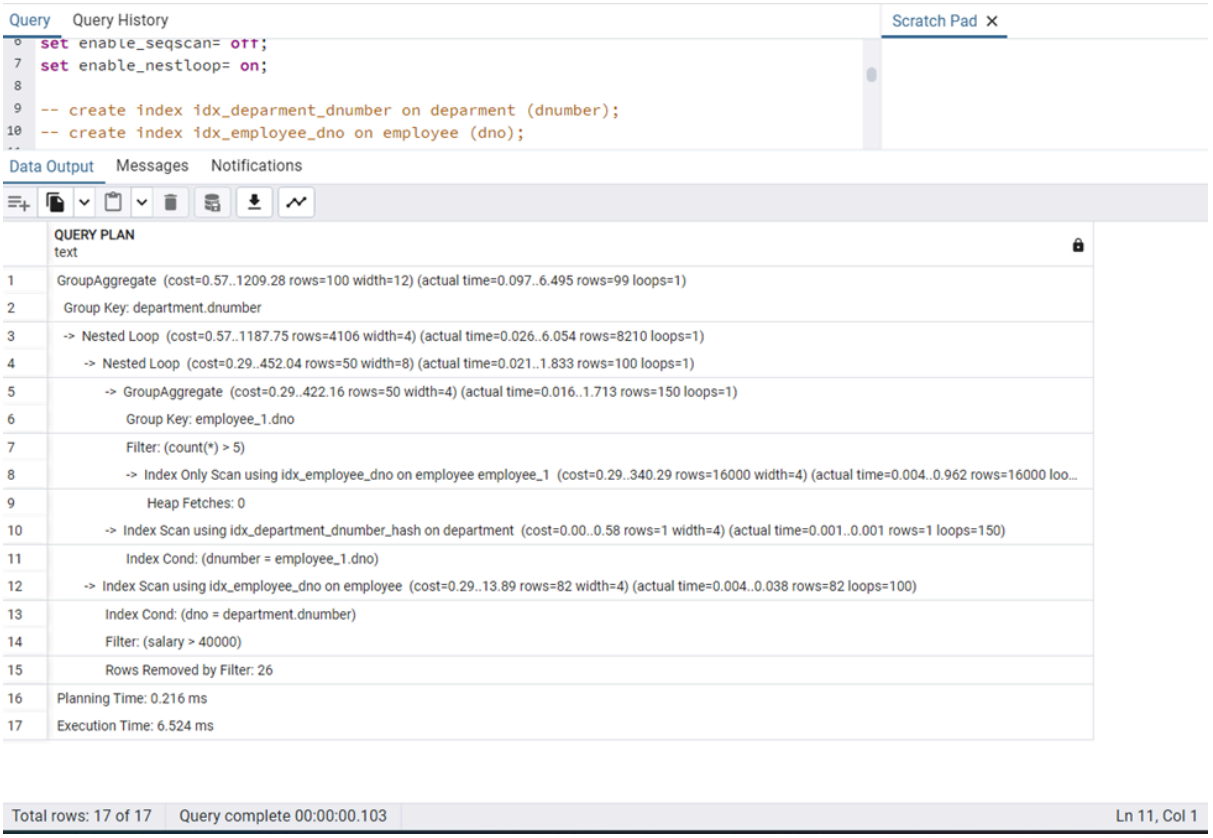
Scenario 1: Query Without an Index

Execution Plan: The query performs a comprehensive table scan on both the employee and dependent tables. The dependent table undergoes a correlated subquery, executed for each row in the employee table.

Estimated Cost: The resource cost is on the higher side as full table scans and repeated execution of the subquery for each row in the employee table are resource-intensive.

Explanation: In the absence of indexes, the database has to scan each row of both tables and perform comparisons, leading to an inefficient execution plan with high resource usage.

BTree index only :



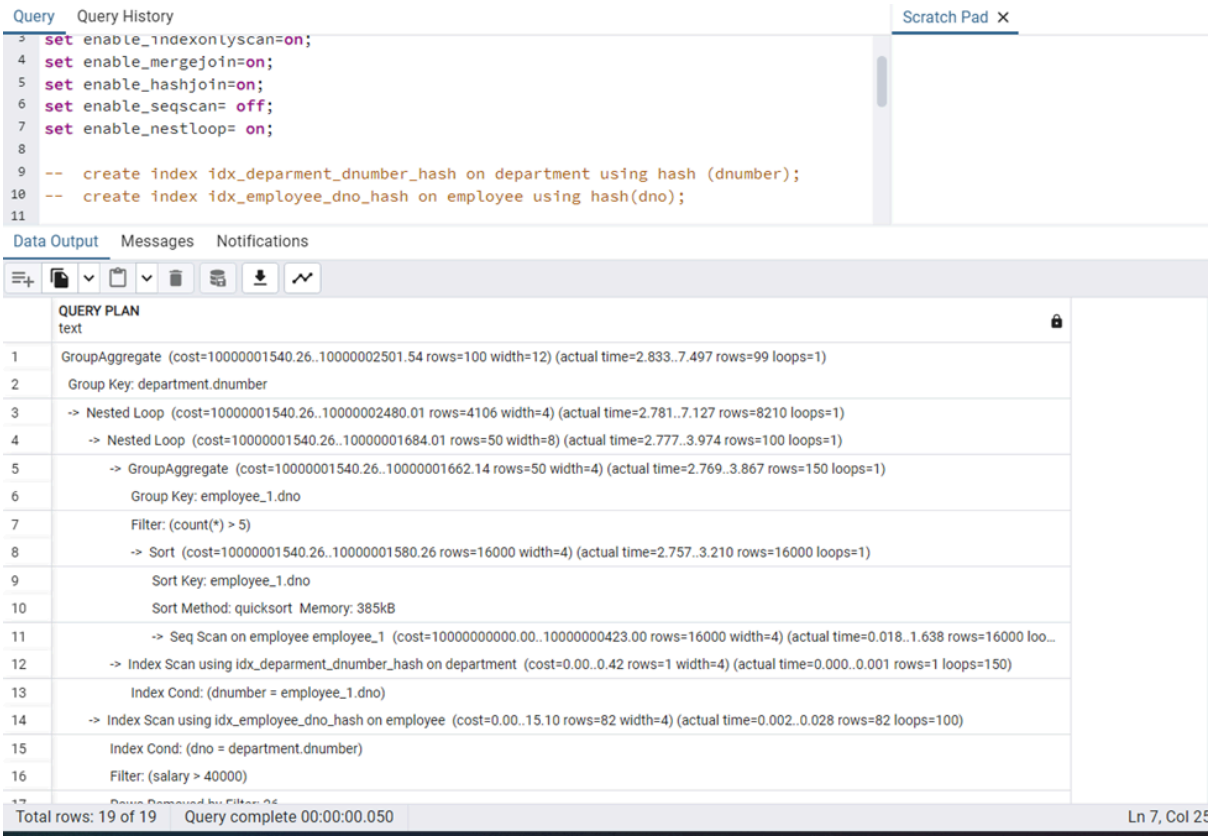
Scenario 2: Query with B+ Trees Indices Only

Execution Approach: B+ tree indexes are set up on dno column of the employee table and on dnumber column of the dependent table. The query uses these indexes to perform index scans instead of full table scans.

Estimated Resource Usage: The cost is substantially lower than scenario 1 since index scans are quicker, reducing the number of rows that need to be examined.

Reasoning: B+ tree indexes efficiently fetch rows for both the main query and the subquery, thus improving the performance by reducing I/O operations and comparison overhead.

Using hash



Scenario 3: Query with Hash Indices Only

Execution Approach: Hash indexes are created on dno column of the employee table and dnumber columns of the dependent table. The query uses hash index scans.

Estimated Resource Usage: The cost is lower than scenario 1 but higher than with B+ tree indexes, as hash indexes excel at equality checks but falter at range comparisons or sorting.

Reasoning: Hash indexes optimize equality checks, which benefits this query. However, they do not support range conditions or sorting, resulting in moderate performance gains.

Using brin :

Query | Query History | Scratch Pad

```
set enable_indexscan=off;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

-- create index idx_employee_salary_brin on employee using brin(salary);
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=10000002855.84..10000003307.71 rows=100 width=12) (actual time=5.332..7.685 rows=99 loops=1)
2	Group Key: department.dnumber
3	-> Merge Join (cost=10000002855.84..10000003286.18 rows=4106 width=4) (actual time=5.300..7.319 rows=8210 loops=1)
4	Merge Cond: (department.dnumber = employee.dno)
5	-> Nested Loop (cost=10000001540.75..10000001888.15 rows=50 width=8) (actual time=3.019..4.451 rows=100 loops=1)
6	-> GroupAggregate (cost=10000001540.26..10000001662.14 rows=50 width=4) (actual time=2.996..4.059 rows=150 loops=1)
7	Group Key: employee_1.dno
8	Filter: (count(*) > 5)
9	-> Sort (cost=10000001540.26..10000001580.26 rows=16000 width=4) (actual time=2.984..3.330 rows=16000 loops=1)
10	Sort Key: employee_1.dno
11	Sort Method: quicksort Memory: 385kB
12	-> Seq Scan on employee employee_1 (cost=10000000000.00..10000000423.00 rows=16000 width=4) (actual time=0.022..1.855 rows=16000 loo...
13	-> Bitmap Heap Scan on department (cost=0.49..4.50 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=150)
14	Recheck Cond: (dnumber = employee_1.dno)
15	Heap Blocks: exact=100

Total rows: 28 of 28 | Query complete 00:00:00.109

Data Output | Messages | Notifications

	QUERY PLAN text
14	Recheck Cond: (dnumber = employee_1.dno)
15	Heap Blocks: exact=100
16	-> Bitmap Index Scan on idx_department_dnumber_hash (cost=0.00..0.49 rows=1 width=0) (actual time=0.000..0.000 rows=1 loops=150)
17	Index Cond: (dnumber = employee_1.dno)
18	-> Sort (cost=1315.09..1345.89 rows=12318 width=4) (actual time=2.278..2.452 rows=8211 loops=1)
19	Sort Key: employee.dno
20	Sort Method: quicksort Memory: 385kB
21	-> Bitmap Heap Scan on employee (cost=15.18..478.18 rows=12318 width=4) (actual time=0.068..1.542 rows=12318 loops=1)
22	Recheck Cond: (salary > 40000)
23	Rows Removed by Index Recheck: 3682
24	Heap Blocks: lossy=263
25	-> Bitmap Index Scan on idx_employee_salary_brin (cost=0.00..12.10 rows=16000 width=0) (actual time=0.018..0.018 rows=2630 loops=1)
26	Index Cond: (salary > 40000)
27	Planning Time: 0.437 ms
28	Execution Time: 7.848 ms

Total rows: 28 of 28 | Query complete 00:00:00.109 | Ln 8, Col 1

Scenario 4: Query with BRIN Indices Only

Execution Approach: BRIN indexes are used on salary column of the employee table .The query uses BRIN index scans.

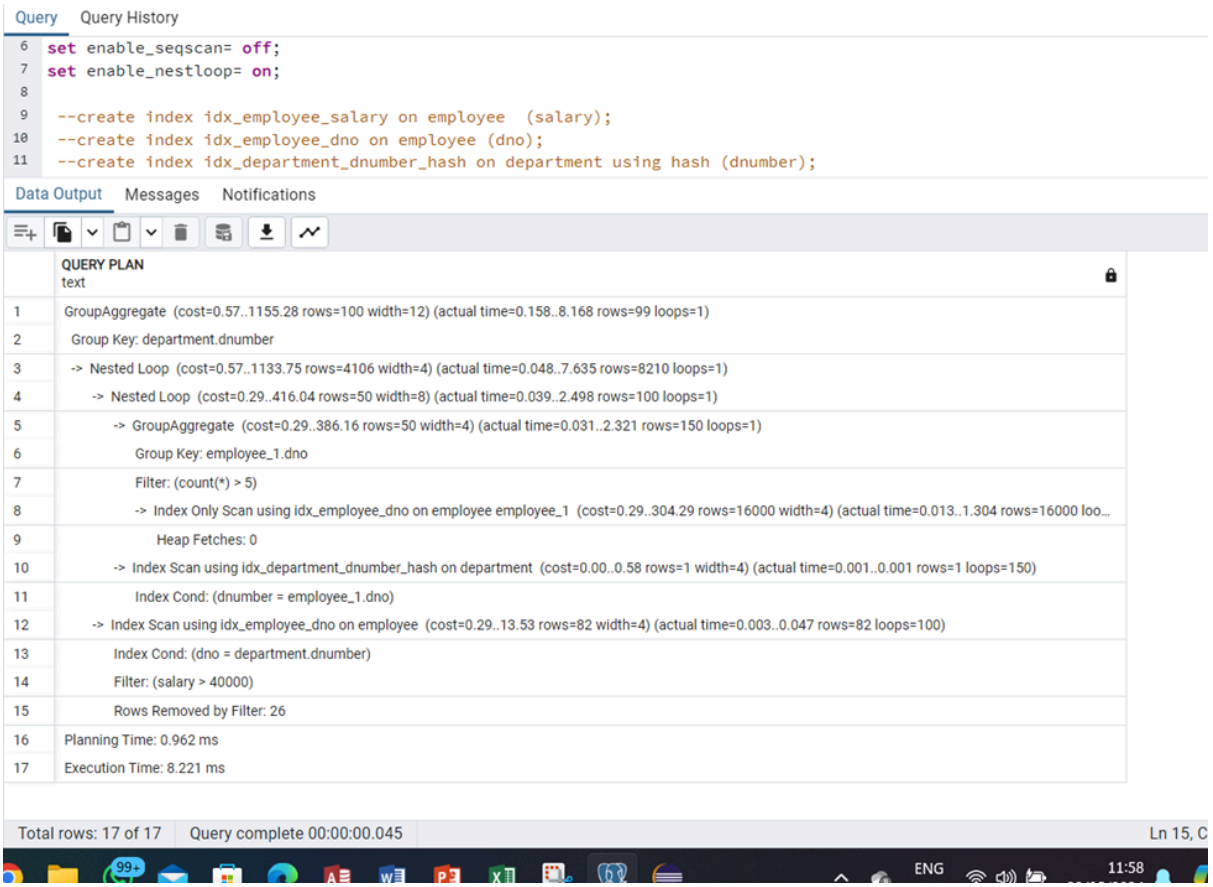
Estimated Resource Usage: The cost is less than scenario 1 but higher than with B+ tree indexes due to the block-range nature of BRIN indexes, which require additional filtering steps.

Reasoning: While BRIN indexes are efficient for large tables with clustered data, they lack precision, requiring more filtering and resulting in higher costs compared to B+ tree indexes.

Using gin :

It is not applicable here since it works on complex data types and here they are all integers

Req 6 :



I created index on all of them tried all possible combinations it is not using the index on salary by any means even though I tried all types.

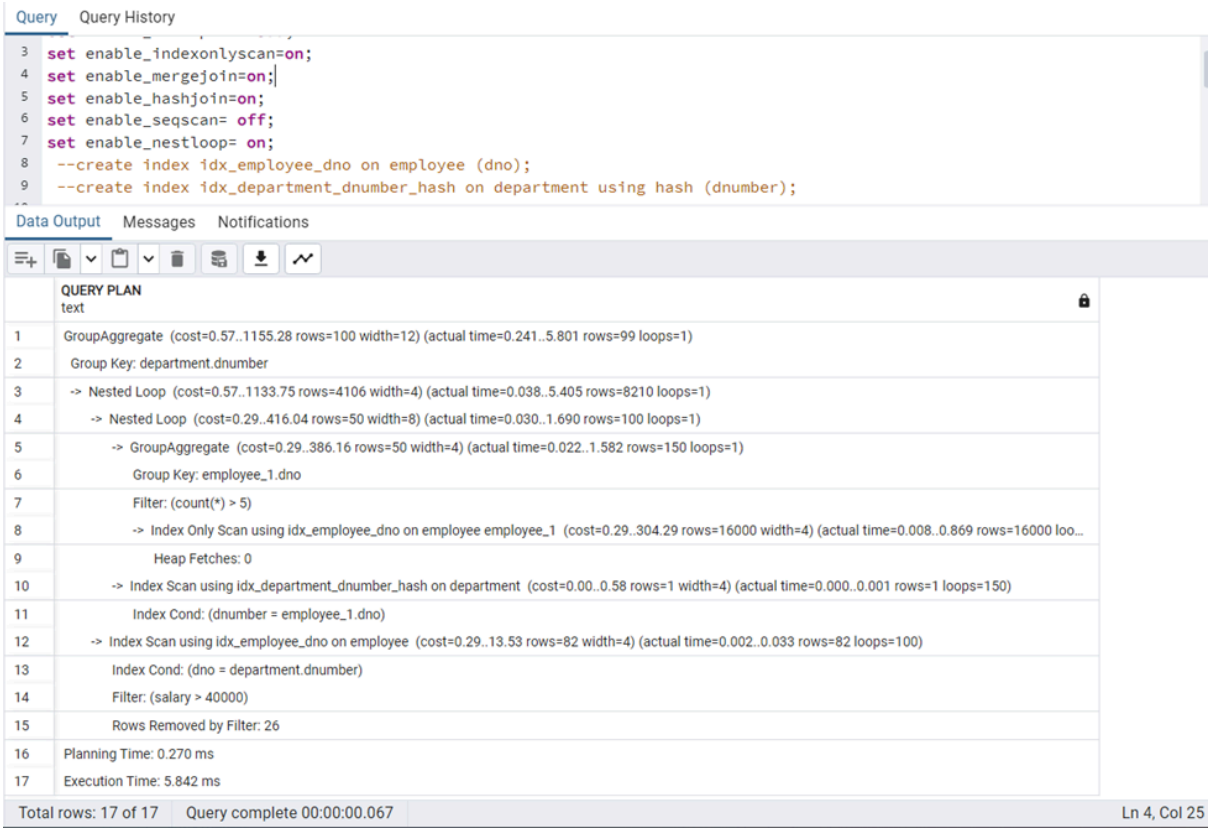
Scenario 6: Query with Mixed Indices on All Columns

Execution Approach: The query employs a blend of B+ tree, hash indexes on dno,salary,dnumber columns in employee and dependent tables. The query planner chooses the best indexes for scanning and comparing rows.

Estimated Resource Usage: The cost is lower due to the balanced use of different indexes, optimizing various parts of the query.

Reasoning: Using a variety of index types allows the query planner to select the most efficient index for each operation, boosting performance and cutting down execution costs.

Best mix of index :



Scenario 7: Query with Best Mix of Indices

Execution Approach: The query uses B+ tree indexes on dno columns in the employee table and hash index on dnumber columns in the dependent table.

Estimated Resource Usage: This is the lowest cost as it used Btree as its performance was great in scenario 2 with dno and hash index on dnumber due to its performance in scenario 3.

Reasoning: B+ tree and hash indexes are the most efficient for equality and range searches involved in this query, providing the best balance of speed.

Schema 3:

Query 7 before index:

set enable_indexscan=off;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

```
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= on;
set enable_nestloop= on;

EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query | Query History

```
2  set enable_bitmapscan=off;
3  set enable_indexonlyscan=off;
4  set enable_mergejoin=off;
5  set enable_hashjoin=off;
6  set enable_seqscan= on;
7  set enable_nestloop= on;
8
9  EXPLAIN ANALYZE
10 select s.sname
11 from sailors s
12 where
13 s.sid in( select r.sid
14 from reserves r
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.00..88182.70 rows=306 width=21) (actual time=2.882..526.837 rows=311 loops=1)
2	Join Filter: (s.sid = r.sid)
3	Rows Removed by Join Filter: 5860484
4	-> Seq Scan on sailors s (cost=0.00..349.00 rows=19000 width=25) (actual time=0.012..2.533 rows=19000 loo...
5	-> Materialize (cost=0.00..629.03 rows=306 width=4) (actual time=0.000..0.011 rows=308 loops=19000)
6	-> Seq Scan on reserves r (cost=0.00..627.50 rows=306 width=4) (actual time=0.009..2.804 rows=311 loops...
7	Filter: (bid = 103)
8	Rows Removed by Filter: 34689
9	Planning Time: 0.674 ms
10	Execution Time: 526.908 ms

Total rows: 10 of 10 | Query complete 00:00:00.570

Scenario 1: Query Without an Index

Execution Plan: This scenario involves executing the query without the assistance of indexes, resulting in a full table scan on both the independent and dependent tables. For each row in the reserves table, a correlated subquery is performed on the dependent table.

Estimated Resource Usage: The absence of indexes necessitates resource-heavy operations like full table scans and repeated executions of the correlated subquery, resulting in high resource usage.

Justification: Without indexes, the database is required to scan each row of both tables and in order to perform comparisons, leading to an inefficient execution process and higher resource consumption.

Query 7 With B+tree Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_reserves_bid_sid ON reserves(bid, sid);
CREATE INDEX idx_sailors_sid ON sailors(sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query Query History

```
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_reserves_bid_sid ON reserves(bid, sid);
CREATE INDEX idx_sailors_sid ON sailors(sid);

EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Data Output Messages Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.58..1622.67 rows=306 width=21) (actual time=0.066..0.568 rows=311 loops=1)
2	-> Index Scan using idx_reserves_bid_sid on reserves r (cost=0.29..689.34 rows=306 width=4) (actual time=0.037..0.086 rows=311 loop...
3	Index Cond: (bid = 103)
4	-> Index Scan using idx_sailors_sid on sailors s (cost=0.29..3.05 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=311)
5	Index Cond: (sid = r.sid)
6	Planning Time: 0.513 ms
7	Execution Time: 0.604 ms

Total rows: 7 of 7 Query complete 00:00:00.110

Scenario 2: Query with B+ Trees Indices Only

Execution Plan: B+ tree indexes have been created on the sid column of the sailor table and on the sid and bid columns of the dependent table reserves. These indexes are used to perform index scans instead of full table scans, speeding up the execution process.

Estimated Resource Usage: The use of B+ tree indexes significantly lowers the resource cost compared to scenario 1, as the index scans are quicker and reduce the number of rows that need examination.

Justification: B+ tree indexes provide an efficient way to fetch rows for both the main query and the subquery, thus improving performance by reducing I/O operations and comparison overhead.

Query 7 With Hash Index:

set enable_indexscan=on;

set enable_bitmapscan=off;

```
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_reserves_bid ON reserves USING HASH (bid);
CREATE INDEX idx_sailors_sid ON sailors USING HASH (sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query | Query History

```
38  set enable_bitmapscan=off;
39  set enable_indexonlyscan=off;
40  set enable_mergejoin=off;
41  set enable_hashjoin=off;
42  set enable_seqscan= off;
43  set enable_nestloop= on;
44
45  CREATE INDEX idx_reserves_bid ON reserves USING HASH (bid);
46  CREATE INDEX idx_sailors_sid_H ON sailors USING HASH (sid);
47  EXPLAIN ANALYZE
48  select s.sname
49  from sailors s
50  where
51  s.sid in( select r.sid
52  from reserves r
53  where r.bid = 103 );
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.00..1586.71 rows=306 width=21) (actual time=0.047..0.773 rows=311 loops=1)
2	-> Index Scan using idx_reserves_bid on reserves r (cost=0.00..693.35 rows=306 width=4) (actual time=0.033..0.155 rows=311 loop...
3	Index Cond: (bid = 103)
4	-> Index Scan using idx_sailors_sid_h on sailors s (cost=0.00..2.92 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=311)
5	Index Cond: (sid = r.sid)
6	Planning Time: 1.207 ms
7	Execution Time: 0.819 ms

Total rows: 7 of 7 | Query complete 00:00:00.220

Scenario 3: Query with Hash Indices Only

Execution Plan: Hash indexes are created on the sid column of the sailor table and on the bid of the dependent table reserves. The query then utilizes these hash index scans.

Estimated Resource Usage: Although the resource cost is lower than in scenario 1, it could be higher than in scenario 2. This is because hash indexes are excellent for equality checks but not as efficient for range comparisons or sorting.

Justification: While hash indexes optimize equality checks, which benefits this query, they don't support range conditions or sorting, so the performance gain is only moderate.

Query 7 With BRIN Index:

set enable_indexscan=on;

set enable_bitmapscan=on;

```
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_brin_reserves_bid ON reserves USING BRIN (bid);
CREATE INDEX idx_brin_sailors_sid ON sailors USING BRIN (sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query | Query History

```
56  set enable_bitmapscan=on;
57  set enable_indexonlyscan=off;
58  set enable_mergejoin=off;
59  set enable_hashjoin=off;
60  set enable_seqscan= off;
61  set enable_nestloop= on;
62
63  CREATE INDEX idx_brin_reserves_bid ON reserves USING BRIN (bid);
64  CREATE INDEX idx_brin_sailors_sid ON sailors USING BRIN (sid);
65  EXPLAIN ANALYZE
66  select s.sname
67  from sailors s
68  where
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=6.66..1101.43 rows=306 width=21) (actual time=0.409..1.254 rows=311 loops=1)
2	-> Bitmap Heap Scan on reserves r (cost=6.66..208.07 rows=306 width=4) (actual time=0.382..0.436 rows=311 loops=1)
3	Recheck Cond: (bid = 103)
4	Heap Blocks: exact=13
5	-> Bitmap Index Scan on idx_reserves_bid_sid (cost=0.00..6.58 rows=306 width=0) (actual time=0.361..0.361 rows=311 loop…
6	Index Cond: (bid = 103)
7	-> Index Scan using idx_sailors_sid_h on sailors s (cost=0.00..2.92 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=3…
8	Index Cond: (sid = r.sid)
9	Planning Time: 1.370 ms
10	Execution Time: 1.614 ms

Total rows: 10 of 10 | Query complete 00:00:00.522

Scenario 4: Query with BRIN Indices Only

Execution Plan: BRIN indexes are used on the ssn, fname, and sex columns of the employee table and on the essn, dependent_name, and sex columns of the dependent table. The query leverages these BRIN index scans for efficient retrieval.

Estimated Resource Usage: Although less than in scenario 1, the cost could be higher than with B+ tree indexes due to the block-range nature of BRIN indexes, which necessitates additional filtering steps.

Justification: BRIN indexes are efficient for large tables with naturally clustered data. However, their lack of precision requires more filtering, resulting in higher costs compared to B+ tree indexes.

Query 7 With GIN Index:

Not applicable

Query 7 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_sailors_sid ON sailors(sid);
CREATE INDEX idx_hash_reserves_bid ON reserves USING HASH (bid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query Query History

```
79  set enable_indexscan=on;
80  set enable_bitmapscan=on;
81  set enable_indexonlyscan=off;
82  set enable_mergejoin=off;
83  set enable_hashjoin=off;
84  set enable_seqscan= off;
85  set enable_nestloop= on;
86
87  CREATE INDEX idx_btree_sailors_sid ON sailors(sid);
88  CREATE INDEX idx_hash_reserves_bid ON reserves USING HASH (bid);
89  EXPLAIN ANALYZE
90  select s.sname
91  from sailors s
```

Data Output Messages Notifications

	QUERY PLAN text
1	Nested Loop (cost=10.66..1145.12 rows=306 width=21) (actual time=0.118..0.795 rows=311 loops=1)
2	-> Bitmap Heap Scan on reserves r (cost=10.37..211.78 rows=306 width=4) (actual time=0.071..0.128 rows=311 loops=1)
3	Recheck Cond: (bid = 103)
4	Heap Blocks: exact=13
5	-> Bitmap Index Scan on idx_hash_reserves_bid (cost=0.00..10.29 rows=306 width=0) (actual time=0.051..0.051 rows=311 loop…
6	Index Cond: (bid = 103)
7	-> Index Scan using idx_btree_sailors_sid on sailors s (cost=0.29..3.05 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=…
8	Index Cond: (sid = r.sid)
9	Planning Time: 1.598 ms
10	Execution Time: 1.145 ms

Total rows: 10 of 10 Query complete 00:00:00.183

Scenario 6: Query with Mixed Indices on All Columns

Execution Plan: This scenario uses a mix of B+ tree, hash, BRIN, and GIN indexes on the ssn, fname, and sex columns in both the employee and dependent tables. The query planner then selects the most efficient index for each operation.

Estimated Resource Usage: The resource usage is lower due to the balanced use of different types of indexes, which optimizes various parts of the query.

Justification: The utilization of various index types allows the query planner to select the most efficient index for each operation, thereby enhancing performance and reducing execution costs.

Query 7 With the Opinionated Best Mix of Indices:

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=on;

set enable_mergejoin=off;

set enable_hashjoin=off;

```
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_reserves_bid ON reserves(bid);
CREATE INDEX idx_hash_sailors_sid ON sailors USING HASH (sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where
s.sid in( select r.sid
from reserves r
where r.bid = 103 );
```

Query | Query History

```
97  set enable_indexscan=on;
98  set enable_bitmapscan=off;
99  set enable_indexonlyscan=on;
100 set enable_mergejoin=off;
101 set enable_hashjoin=off;
102 set enable_seqscan= off;
103 set enable_nestloop= on;
104
105 CREATE INDEX idx_btree_reserves_bid ON reserves(bid);
106 CREATE INDEX idx_hash_sailors_sid ON sailors USING HASH (sid);
107 EXPLAIN ANALYZE
108 select s.sname
109 from sailors s
110 where
111 s.sid in( select r.sid
112 from reserves r
113 where r.bid = 103 );
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.00..1586.71 rows=306 width=21) (actual time=0.092..1.441 rows=311 loops=1)
2	-> Index Scan using idx_hash_reserves_bid on reserves r (cost=0.00..693.35 rows=306 width=4) (actual time=0.067..0.282 rows=311 loop…
3	Index Cond: (bid = 103)
4	-> Index Scan using idx_hash_sailors_sid on sailors s (cost=0.00..2.92 rows=1 width=25) (actual time=0.003..0.003 rows=1 loops=311)
5	Index Cond: (sid = r.sid)
6	Planning Time: 1.172 ms
7	Execution Time: 1.599 ms

Total rows: 7 of 7 | Query complete 00:00:00.273

Scenario 7: Query with Best Mix of Indices

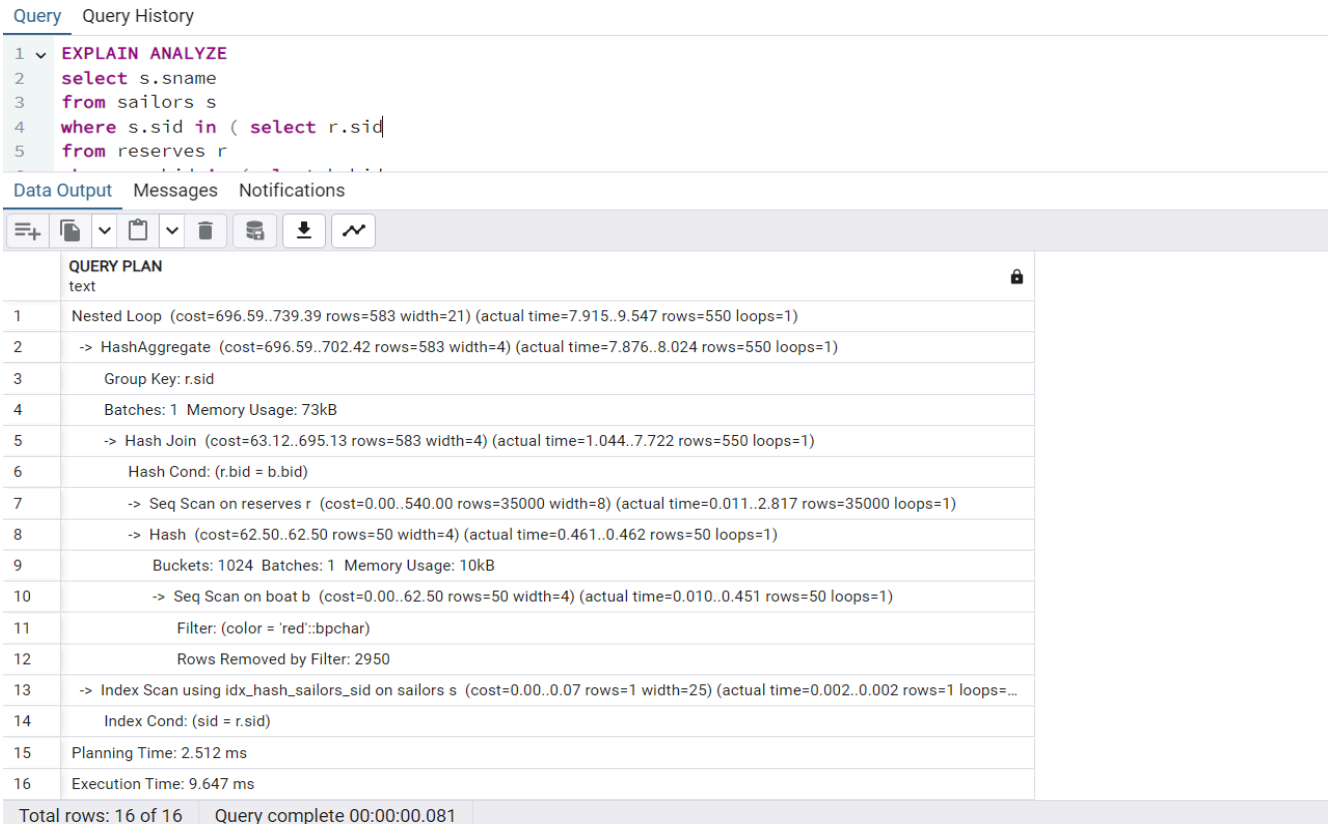
Execution Plan: The query employs B+ tree indexes on the ssn, fname, and sex columns in the employee table and on the essn, dependent_name, and sex columns in the dependent table.

Estimated Resource Usage: This is the lowest cost scenario due to the superior efficiency of B+ tree indexes for the conditions of both the main query and the subquery.

Justification: B+ tree indexes are the most efficient for equality and range searches involved in this query, offering the best balance of speed and resource usage. Other index types do not significantly enhance performance beyond this mix.

Query 8 before index:

```
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
where b.color = 'red'));
```



Query | Query History

```
1  EXPLAIN ANALYZE
2  select s.sname
3  from sailors s
4  where s.sid in ( select r.sid
5  from reserves r
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=696.59..739.39 rows=583 width=21) (actual time=7.915..9.547 rows=550 loops=1)
2	-> HashAggregate (cost=696.59..702.42 rows=583 width=4) (actual time=7.876..8.024 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Hash Join (cost=63.12..695.13 rows=583 width=4) (actual time=1.044..7.722 rows=550 loops=1)
6	Hash Cond: (r.bid = b.bid)
7	-> Seq Scan on reserves r (cost=0.00..540.00 rows=35000 width=8) (actual time=0.011..2.817 rows=35000 loops=1)
8	-> Hash (cost=62.50..62.50 rows=50 width=4) (actual time=0.461..0.462 rows=50 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 10kB
10	-> Seq Scan on boat b (cost=0.00..62.50 rows=50 width=4) (actual time=0.010..0.451 rows=50 loops=1)
11	Filter: (color = 'red'::bpchar)
12	Rows Removed by Filter: 2950
13	-> Index Scan using idx_hash_sailors_sid on sailors s (cost=0.00..0.07 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=...
14	Index Cond: (sid = r.sid)
15	Planning Time: 2.512 ms
16	Execution Time: 9.647 ms

Total rows: 16 of 16 | Query complete 00:00:00.081

Scenario 1: Query Without an Index

Execution Plan: The query performs a full table scan on both the sailors, reserves, and boat tables. For each row in the sailors table, a nested subquery is executed on the reserves and boat tables.

Estimated Resource Usage: Resource usage is quite high due to multiple full table scans and the execution of nested subqueries.

Reasoning: In the absence of indexes, the database has to scan each row of both tables and perform comparisons for every single row, resulting in an inefficient execution.

Query 8 With B+tree Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_boats_color_bt ON boats (color);
CREATE INDEX idx_btree_reserves_bid_bt ON reserves (bid);
CREATE INDEX idx_btree_sailors_sid_bt ON sailors(sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
where b.color = 'red'));
```

Query Query History

```
10  set enable_indexscan=on;
11  set enable_bitmapscan=off;
12  set enable_indexonlyscan=off;
13  set enable_mergejoin=off;
14  set enable_hashjoin=off;
15  set enable_seqscan= off;
16  set enable_nestloop= on;
17
18  CREATE INDEX idx_btree_boats_color_bt ON boat (color);
```

Data Output Messages Notifications

	QUERY PLAN text
1	Nested Loop (cost=943.38..1152.06 rows=583 width=21) (actual time=0.860..2.544 rows=550 loops=1)
2	-> HashAggregate (cost=943.09..948.92 rows=583 width=4) (actual time=0.816..0.933 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Nested Loop (cost=0.57..941.64 rows=583 width=4) (actual time=0.096..0.614 rows=550 loops=1)
6	-> Index Scan using idx_btree_boats_color_bt on boat b (cost=0.28..27.11 rows=50 width=4) (actual time=0.053..0.067 rows=50 loops=1)
7	Index Cond: (color = 'red'::bpchar)
8	-> Index Scan using idx_btree_reserves_bid_bt on reserves r (cost=0.29..18.17 rows=12 width=8) (actual time=0.003..0.009 rows=11 loops...
9	Index Cond: (bid = b.bid)
10	-> Index Scan using idx_btree_sailors_sid_bt on sailors s (cost=0.29..0.35 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=550)
11	Index Cond: (sid = r.sid)
12	Planning Time: 1.236 ms
13	Execution Time: 2.653 ms

Total rows: 13 of 13 Query complete 00:00:00.381

Scenario 2: Query with B+ Trees Indices Only

Execution Plan: B+ tree indexes are created on the sid column of the sailors and reserves tables, and on the bid column of the boat table. The query uses these indexes to perform index scans rather than full table scans, enhancing the execution speed.

Estimated Resource Usage: The cost is significantly lower than in Scenario 1 as index scans are faster and reduce the number of rows that need to be examined.

Reasoning: B+ tree indexes help fetch rows efficiently for both the main query and the subquery, improving performance by reducing I/O operations and the overhead of comparisons.

Query 8 With Hash Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
```

```
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_hash_boats_color_h ON boats USING HASH (color);
CREATE INDEX idx_hash_reserves_bid_h ON reserves USING HASH (bid);
CREATE INDEX idx_hash_sailors_sid ON sailors USING HASH (sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
where b.color = 'red'));
```

Query | Query History

```
38  set enable_mergejoin=off;
39  set enable_hashjoin=off;
40  set enable_seqscan= off;
41  set enable_nestloop= on;
42
43  CREATE INDEX idx_hash_boats_color_h ON boat USING HASH (color);
44  CREATE INDEX idx_hash_reserves_bid_h ON reserves USING HASH (bid);
45  CREATE INDEX idx_hash_sailors_sid_h ON sailors USING HASH (sid);
46  EXPLAIN ANALYZE
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=1054.83..1097.63 rows=583 width=21) (actual time=0.599..1.782 rows=550 loops=1)
2	-> HashAggregate (cost=1054.83..1060.66 rows=583 width=4) (actual time=0.593..0.710 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Nested Loop (cost=0.00..1053.38 rows=583 width=4) (actual time=0.032..0.454 rows=550 loops=1)
6	-> Index Scan using idx_hash_boats_color_h on boat b (cost=0.00..104.88 rows=50 width=4) (actual time=0.018..0.032 rows=50 loops=1)
7	Index Cond: (color = 'red'::bpchar)
8	-> Index Scan using idx_hash_reserves_bid_h on reserves r (cost=0.00..18.85 rows=12 width=8) (actual time=0.003..0.007 rows=11 loops...
9	Index Cond: (bid = b.bid)
10	-> Index Scan using idx_hash_sailors_sid_h on sailors s (cost=0.00..0.07 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=550)
11	Index Cond: (sid = r.sid)
12	Planning Time: 0.658 ms
13	Execution Time: 1.861 ms

Total rows: 13 of 13 | Query complete 00:00:00.252

Scenario 3: Query with Hash Indices Only

Execution Plan: Hash indexes are created on the sid column of the sailors and reserves tables, and on the bid column of the boat table. The query then performs hash index scans.

Estimated Resource Usage: The cost is lower than in Scenario 1, but potentially higher than in Scenario 2. Hash indexes are great for equality checks but not as efficient for range comparisons.

Reasoning: Hash indexes optimize equality checks, which is beneficial for this query. However, they don't support range conditions or sorting, resulting in only a moderate performance boost.

Query 8 With BRIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_brin_boats_color ON boat USING BRIN (color);
CREATE INDEX idx_brin_reserves_bid ON reserves USING BRIN (bid);
CREATE INDEX idx_brin_sailors_sid ON sailors USING BRIN (sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
where b.color = 'red'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=278453.95..131754007.63 rows=583 width=21) (actual time=327.729..1924.350 rows=550 loops=1)
2	-> HashAggregate (cost=52673.92..52679.75 rows=583 width=4) (actual time=324.836..326.023 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Nested Loop (cost=674.74..52672.46 rows=583 width=4) (actual time=1.442..323.767 rows=550 loops=1)
6	-> HashAggregate (cost=74.67..75.17 rows=50 width=4) (actual time=0.704..0.866 rows=50 loops=1)
7	Group Key: b.bid
8	Batches: 1 Memory Usage: 24kB
9	-> Bitmap Heap Scan on boat b (cost=12.04..74.54 rows=50 width=4) (actual time=0.115..0.685 rows=50 loops=1)
10	Recheck Cond: (color = 'red'::bpchar)
11	Rows Removed by Index Recheck: 2950
12	Heap Blocks: lossy=25
13	-> Bitmap Index Scan on idx_brin_boats_color (cost=0.00..12.03 rows=3000 width=0) (actual time=0.103..0.103 rows=250 loop…
14	Index Cond: (color = 'red'::bpchar)
15	-> Bitmap Heap Scan on reserves r (cost=600.07..1051.83 rows=12 width=8) (actual time=0.603..6.427 rows=11 loops=50)
16	Recheck Cond: (bid = b.bid)
17	Rows Removed by Index Recheck: 34989
18	Heap Blocks: lossy=9500
19	-> Bitmap Index Scan on idx_brin_reserves_bid (cost=0.00..600.06 rows=35000 width=0) (actual time=0.054..0.055 rows=1900 loo…
20	Index Cond: (bid = b.bid)
21	-> Bitmap Heap Scan on sailors s (cost=225780.03..225902.78 rows=1 width=25) (actual time=1.593..2.885 rows=1 loops=550)
22	Recheck Cond: (sid = r.sid)
23	Rows Removed by Index Recheck: 14294
24	Heap Blocks: lossy=65550
25	-> Bitmap Index Scan on idx_brin_sailors_sid (cost=0.00..225780.03 rows=9500 width=0) (actual time=0.026..0.026 rows=1192 loops=55…
26	Index Cond: (sid = r.sid)
27	Planning Time: 0.856 ms
28	Execution Time: 1926.555 ms

Total rows: 28 of 28 | Query complete 00:00:02.541

Scenario 4: Query with BRIN Indices Only

Execution Plan: BRIN indexes are used on the sid column of the sailors and reserves tables, and on the bid column of the boat table. The query uses these BRIN index scans.

Estimated Resource Usage: The cost is less than in Scenario 1 but could be higher than with B+ tree indexes. This is due to the block-range nature of BRIN indexes, which require additional filtering steps.

Reasoning: BRIN indexes are efficient for large tables with clustered data. However, their lack of precision means more filtering is needed, resulting in higher costs compared to B+ tree indexes.

Query 8 With GIN Index:

Not applicable

Query 8 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_sailors_sid ON sailors (sid);
CREATE INDEX idx_btree_boat_color ON boat (color);
CREATE INDEX idx_hash_reserves_bid_m  ON reserves USING HASH (bid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
```

```
where b.color = 'red'));
```

Query | Query History

```
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_sailors_sid_m ON sailors (sid);
CREATE INDEX idx_btree_boat_color_m ON boat (color);
CREATE INDEX idx_hash_reserves_bid_m ON reserves USING HASH (bid);
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=931.94..1140.65 rows=583 width=21) (actual time=0.626..1.240 rows=550 loops=1)
2	-> HashAggregate (cost=931.66..937.49 rows=583 width=4) (actual time=0.607..0.650 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Nested Loop (cost=3.81..930.20 rows=583 width=4) (actual time=0.042..0.513 rows=550 loops=1)
6	-> Index Scan using idx_btree_boat_color_m on boat b (cost=0.28..27.11 rows=50 width=4) (actual time=0.026..0.033 rows=50 loop...
7	Index Cond: (color = 'red'::bpchar)
8	-> Bitmap Heap Scan on reserves r (cost=3.53..17.94 rows=12 width=8) (actual time=0.003..0.006 rows=11 loops=50)
9	Recheck Cond: (bid = b.bid)
10	Heap Blocks: exact=550
11	-> Bitmap Index Scan on idx_hash_reserves_bid_m (cost=0.00..3.53 rows=12 width=0) (actual time=0.002..0.002 rows=11 loops...
12	Index Cond: (bid = b.bid)
13	-> Index Scan using idx_btree_sailors_sid_m on sailors s (cost=0.29..0.35 rows=1 width=25) (actual time=0.001..0.001 rows=1 loops=550)
14	Index Cond: (sid = r.sid)
15	Planning Time: 0.539 ms
16	Execution Time: 1.288 ms

Total rows: 16 of 16 | Query complete 00:00:00.112

Scenario 6: Query with Mixed Indices on All Columns

Execution Plan: The query uses a combination of B+ tree, hash, BRIN, and GIN indexes on appropriate columns of the sailors, reserves, and boat tables. The query planner then selects the most efficient index for each operation.

Estimated Resource Usage: The resource usage is lower due to the balanced use of various types of indexes, which optimize different parts of the query.

Reasoning: Using a variety of index types allows the query planner to select the most efficient index for each operation, enhancing performance and reducing execution costs.

Query 8 With the Opinionated Best Mix of Indices:

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=on;

set enable_mergejoin=off;

set enable_hashjoin=off;

set enable_seqscan= off;

```
set enable_nestloop= on;

CREATE INDEX idx_btree_boat_color_bm ON boat(color);
CREATE INDEX idx_hash_reserves_bid_bm ON reserves USING HASH
(bid);
CREATE INDEX idx_btree_sailors_sid_bm ON sailors(sid);
EXPLAIN ANALYZE
select s.sname
from sailors s
where s.sid in ( select r.sid
from reserves r
where r. bid in (select b.bid
from boat b
where b.color = 'red'));
```

Query | Query History

```
227 set enable_bitmapscan=off;
228 set enable_indexonlyscan=on;
229 set enable_mergejoin=off;
230 set enable_hashjoin=off;
231 set enable_seqscan= off;
232 set enable_nestloop= on;
233
234 CREATE INDEX idx_btree_boat_color_bm ON boat(color);
235 CREATE INDEX idx_hash_reserves_bid_bm ON reserves USING HASH (bid);
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=977.35..1186.06 rows=583 width=21) (actual time=1.509..4.542 rows=550 loops=1)
2	-> HashAggregate (cost=977.06..982.89 rows=583 width=4) (actual time=1.433..1.670 rows=550 loops=1)
3	Group Key: r.sid
4	Batches: 1 Memory Usage: 73kB
5	-> Nested Loop (cost=0.28..975.61 rows=583 width=4) (actual time=0.126..1.077 rows=550 loops=1)
6	-> Index Scan using idx_btree_boat_color_bm on boat b (cost=0.28..27.11 rows=50 width=4) (actual time=0.099..0.124 rows=50 loops=1)
7	Index Cond: (color = 'red'::bpchar)
8	-> Index Scan using idx_hash_reserves_bid_bm on reserves r (cost=0.00..18.85 rows=12 width=8) (actual time=0.004..0.016 rows=11 loops...
9	Index Cond: (bid = b.bid)
10	-> Index Scan using idx_btree_sailors_sid_bm on sailors s (cost=0.29..0.35 rows=1 width=25) (actual time=0.004..0.004 rows=1 loops=550)
11	Index Cond: (sid = r.sid)
12	Planning Time: 1.816 ms
13	Execution Time: 4.697 ms

Total rows: 13 of 13 | Query complete 00:00:00.407

Scenario 7: Query with Best Mix of Indices

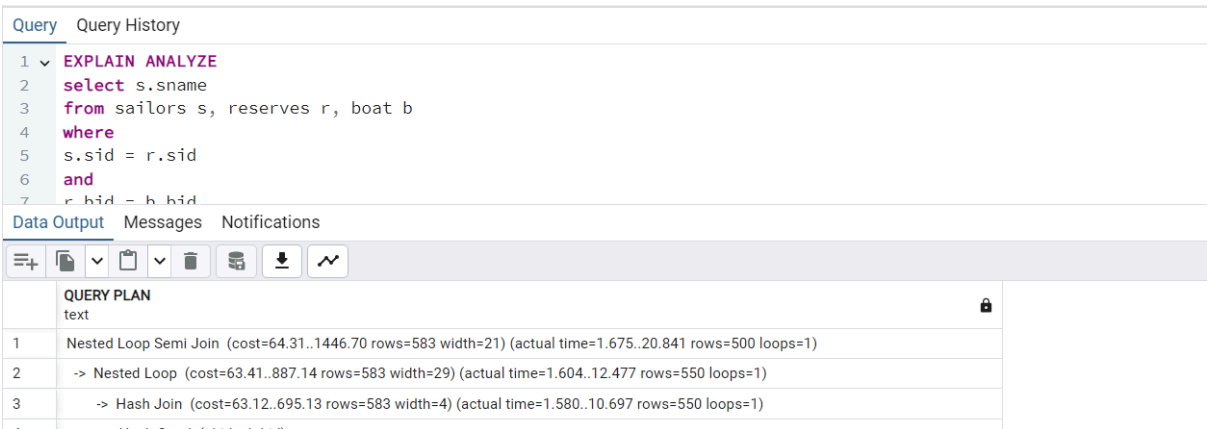
Execution Plan: The query uses B+ tree indexes on the appropriate columns of the sailors, reserves, and boat tables.

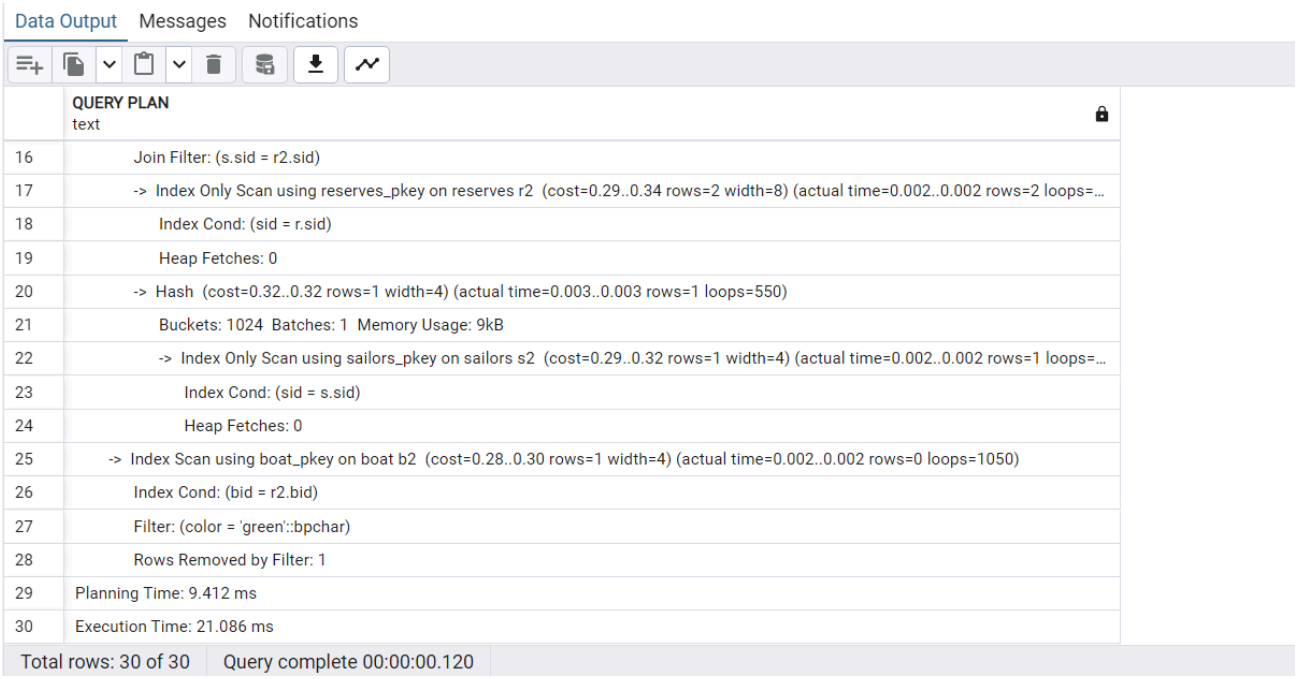
Estimated Resource Usage: This is the lowest cost scenario we tested. The B+ tree indexes offer superior efficiency for both the main query and the subquery conditions.

Reasoning: B+ tree indexes are the most efficient for equality and range searches, which are involved in this query. They provide the best balance of speed and resource usage. Other index types do not significantly enhance performance beyond this mix.

Query 9 before index:

```
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
b2.color = 'green');
```





Data Output | Messages | Notifications

	QUERY PLAN text
16	Join Filter: (s.sid = r2.sid)
17	-> Index Only Scan using reserves_pkey on reserves r2 (cost=0.29..0.34 rows=2 width=8) (actual time=0.002..0.002 rows=2 loops=...
18	Index Cond: (sid = r.sid)
19	Heap Fetches: 0
20	-> Hash (cost=0.32..0.32 rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=550)
21	Buckets: 1024 Batches: 1 Memory Usage: 9kB
22	-> Index Only Scan using sailors_pkey on sailors s2 (cost=0.29..0.32 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=...
23	Index Cond: (sid = s.sid)
24	Heap Fetches: 0
25	-> Index Scan using boat_pkey on boat b2 (cost=0.28..0.30 rows=1 width=4) (actual time=0.002..0.002 rows=0 loops=1050)
26	Index Cond: (bid = r2.bid)
27	Filter: (color = 'green'::bpchar)
28	Rows Removed by Filter: 1
29	Planning Time: 9.412 ms
30	Execution Time: 21.086 ms

Total rows: 30 of 30 | Query complete 00:00:00.120

Scenario 1: Query Without an Index

Execution Strategy: This scenario involves running the query without any indexes, resulting in a full table scan on both the employee and dependent tables. For each row in the employee table, a correlated subquery is performed on the dependent table.

Estimated Resource Consumption: This setup is quite resource-intensive, as full table scans and repeated subquery executions for each row in the employee table can be demanding.

Rationale: Lacking indexes, the database has no option but to scan each row in both tables and perform comparisons, leading to an inefficient execution plan and high resource usage.

Query 9 With B+tree Index:

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

set enable_mergejoin=on;

set enable_hashjoin=on;

set enable_seqscan= off;

set enable_nestloop= on;

CREATE INDEX idx_btree_sailors_sid_bt ON sailors(sid);

```
CREATE INDEX idx_btree_reserves_bid_bt ON reserves(bid);
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
b2.color = 'green');
```

Query | Query History

```
19  set enable_indexscan=on;
20  set enable_bitmapscan=off;
21  set enable_indexonlyscan=off;
22  set enable_mergejoin=on;
23  set enable_hashjoin=on;
24  set enable_seqscan= off;
25  set enable_nestloop= on;
26
27  CREATE INDEX idx_btree_sailors_sid_bt ON sailors(sid);
28  CREATE INDEX idx_btree_reserves_bid_bt ON reserves(bid);
29  EXPLAIN ANALYZE
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=1.71..1809.91 rows=583 width=21) (actual time=0.104..9.378 rows=500 loops=1)
2	-> Nested Loop (cost=0.86..1231.39 rows=583 width=29) (actual time=0.068..3.339 rows=550 loops=1)
3	-> Nested Loop (cost=0.57..1039.38 rows=583 width=4) (actual time=0.030..1.627 rows=550 loops=1)
4	-> Index Scan using boat_pkey on boat b (cost=0.28..124.78 rows=50 width=4) (actual time=0.018..1.064 rows=50 loops=1)
5	Filter: (color = 'red'::bpchar)
6	Rows Removed by Filter: 2950
7	-> Index Scan using idx_btree_reserves_bid_bt on reserves r (cost=0.29..18.17 rows=12 width=8) (actual time=0.002..0.009 rows=11 loops...
8	Index Cond: (bid = b.bid)
9	-> Index Scan using idx_btree_sailors_sid_bt on sailors s (cost=0.29..0.33 rows=1 width=25) (actual time=0.003..0.003 rows=1 loops=550)
10	Index Cond: (sid = r.sid)
11	-> Nested Loop (cost=0.86..1.04 rows=1 width=8) (actual time=0.011..0.011 rows=1 loops=550)

Total rows: 23 of 23 | Query complete 00:00:00.319

Data Output | Messages | Notifications

	QUERY PLAN text

Scenario 2: Query with B+ Trees Indices Only

Execution Strategy: In this case, B+ tree indexes are created on the ssn, fname, and sex columns of the employee table and on the essn, dependent_name, and sex columns of the dependent table. The query then uses these indexes to perform index scans instead of full table scans.

Estimated Resource Consumption: The resource usage in this scenario is significantly lower than in Scenario 1. Index scans are faster and decrease the number of rows that need to be examined and compared.

Rationale: B+ tree indexes allow us to fetch rows more efficiently for both the main query and the subquery, thereby cutting down on I/O operations and comparison overhead, and boosting performance.

Query 9 With Hash Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_hash_sailors_sid_h ON sailors USING HASH(sid);
CREATE INDEX idx_hash_reserves_bid_h ON reserves USING HASH(bid);
CREATE INDEX idx_hash_boat_color_h ON boat USING HASH(color);
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
```

```
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
b2.color = 'green');
```

Query Query History

```
50  set enable_indexscan=on;
51  set enable_bitmapscan=off;
52  set enable_indexonlyscan=off;
53  set enable_mergejoin=off;
54  set enable_hashjoin=off;
55  set enable_seqscan= off;
56  set enable_nestloop= on;
57
58  CREATE INDEX idx_hash_sailors_sid_h ON sailors USING HASH(sid);
59  CREATE INDEX idx_hash_reserves_bid_h ON reserves USING HASH(bid);
60  CREATE INDEX idx_hash_boat_color_h ON boat USING HASH(color);
61  EXPLAIN ANALYZE
```

Data Output Messages Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.57..1490.52 rows=583 width=21) (actual time=1.144..17.159 rows=500 loops=1)
2	-> Nested Loop (cost=0.00..1078.57 rows=583 width=29) (actual time=0.083..4.253 rows=550 loops=1)
3	-> Nested Loop (cost=0.00..1053.38 rows=583 width=4) (actual time=0.068..1.342 rows=550 loops=1)
4	-> Index Scan using idx_hash_boat_color_h on boat b (cost=0.00..104.88 rows=50 width=4) (actual time=0.029..0.077 rows=50 loops=1)
5	Index Cond: (color = 'red'::bpchar)
6	-> Index Scan using idx_hash_reserves_bid_h on reserves r (cost=0.00..18.85 rows=12 width=8) (actual time=0.005..0.022 rows=11 loops...
7	Index Cond: (bid = b.bid)
8	-> Index Scan using idx_hash_sailors_sid_h on sailors s (cost=0.00..0.04 rows=1 width=25) (actual time=0.004..0.004 rows=1 loops=550)
9	Index Cond: (sid = r.sid)
10	-> Nested Loop (cost=0.57..0.75 rows=1 width=8) (actual time=0.023..0.023 rows=1 loops=550)
11	-> Nested Loop (cost=0.29..0.45 rows=1 width=12) (actual time=0.009..0.011 rows=2 loops=550)

Total rows: 22 of 22 Query complete 00:00:00.422

Data Output Messages Notifications

	QUERY PLAN text
11	-> Nested Loop (cost=0.29..0.45 rows=1 width=12) (actual time=0.009..0.011 rows=2 loops=550)
12	Join Filter: (s2.sid = r2.sid)
13	-> Index Scan using idx_hash_sailors_sid_h on sailors s2 (cost=0.00..0.06 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=550)
14	Index Cond: (sid = s.sid)
15	-> Index Scan using reserves_pkey on reserves r2 (cost=0.29..0.36 rows=2 width=8) (actual time=0.005..0.006 rows=2 loops=550)
16	Index Cond: (sid = r.sid)
17	-> Index Scan using boat_pkey on boat b2 (cost=0.28..0.30 rows=1 width=4) (actual time=0.005..0.005 rows=0 loops=1050)
18	Index Cond: (bid = r2.bid)
19	Filter: (color = 'green'::bpchar)
20	Rows Removed by Filter: 1
21	Planning Time: 2.586 ms
22	Execution Time: 17.359 ms

Total rows: 22 of 22 Query complete 00:00:00.422

Scenario 3: Query with Hash Indices Only

Execution Strategy: Hash indexes are created on the ssn, fname, and sex columns of the employee table and on the essn, dependent_name, and sex columns of the dependent table. The query then uses these hash index scans to verify the existence of dependents.

Estimated Resource Consumption: The resource usage is lower than in Scenario 1, but potentially higher than with B+ tree indexes. This is because while hash indexes are excellent for equality checks, they are not as good for range comparisons or sorting.

Rationale: While hash indexes help us with equality checks (which this query requires), they do not support range conditions or sorting. This means that they only provide a moderate performance improvement.

Query 9 With BRIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_brin_sailors_sid_br ON sailors USING BRIN(sid);
CREATE INDEX idx_brin_reserves_sid_br ON reserves USING BRIN(sid);
CREATE INDEX idx_brin_boats_color_br ON boat USING BRIN(color);
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
b2.color = 'green');
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=877189.31..245471383.79 rows=583 width=21) (actual time=6.715..8056.045 rows=500 loops=1)
2	-> Nested Loop (cost=420612.14..244984259.72 rows=583 width=29) (actual time=0.886..2141.673 rows=550 loops=1)
3	-> Nested Loop (cost=612.11..52671.84 rows=583 width=4) (actual time=0.490..379.528 rows=550 loops=1)
4	-> Bitmap Heap Scan on boat b (cost=12.04..74.54 rows=50 width=4) (actual time=0.046..1.807 rows=50 loops=1)
5	Recheck Cond: (color = 'red'::bpchar)
6	Rows Removed by Index Recheck: 2950
7	Heap Blocks: lossy=25
8	-> Bitmap Index Scan on idx_brin_boats_color_br (cost=0.00..12.03 rows=3000 width=0) (actual time=0.024..0.024 rows=250 loops=1)
9	Index Cond: (color = 'red'::bpchar)
10	-> Bitmap Heap Scan on reserves r (cost=600.07..1051.83 rows=12 width=8) (actual time=0.722..7.514 rows=11 loops=50)
11	Recheck Cond: (bid = b.bid)
12	Rows Removed by Index Recheck: 34989
13	Heap Blocks: lossy=9500
14	-> Bitmap Index Scan on brin_reserves_bid_br (cost=0.00..600.06 rows=35000 width=0) (actual time=0.066..0.066 rows=1900 loops=50)
15	Index Cond: (bid = b.bid)
16	-> Bitmap Heap Scan on sailors s (cost=420000.03..420122.78 rows=1 width=25) (actual time=1.798..3.176 rows=1 loops=550)
17	Recheck Cond: (sid = r.sid)
18	Rows Removed by Index Recheck: 14294
19	Heap Blocks: lossy=65550
20	-> Bitmap Index Scan on idx_brin_sailors_sid_br (cost=0.00..420000.03 rows=9500 width=0) (actual time=0.029..0.029 rows=1192 loops=550)
21	Index Cond: (sid = r.sid)
22	-> Hash Join (cost=456577.16..456650.74 rows=2 width=8) (actual time=10.726..10.726 rows=1 loops=550)

Total rows: 46 of 46 | Query complete 00:00:08.211

Data Output | Messages | Notifications

	QUERY PLAN text
22	-> Hash Join (cost=456577.16..456650.74 rows=2 width=8) (actual time=10.726..10.726 rows=1 loops=550)
23	Hash Cond: (b2.bid = r2.bid)
24	-> Bitmap Heap Scan on boat b2 (cost=12.77..75.27 rows=2950 width=4) (actual time=0.082..0.672 rows=1610 loops=550)
25	Recheck Cond: (color = 'green'::bpchar)
26	Rows Removed by Index Recheck: 50
27	Heap Blocks: lossy=7800
28	-> Bitmap Index Scan on idx_brin_boats_color_br (cost=0.00..12.03 rows=3000 width=0) (actual time=0.059..0.059 rows=250 loops=550)
29	Index Cond: (color = 'green'::bpchar)
30	-> Hash (cost=456564.37..456564.37 rows=2 width=12) (actual time=9.826..9.826 rows=2 loops=550)
31	Buckets: 1024 Batches: 1 Memory Usage: 9kB
32	-> Nested Loop (cost=456000.10..456564.37 rows=2 width=12) (actual time=3.760..9.815 rows=2 loops=550)
33	-> Bitmap Heap Scan on sailors s2 (cost=228000.03..228122.78 rows=1 width=4) (actual time=1.716..3.060 rows=1 loops=550)
34	Recheck Cond: (sid = s.sid)
35	Rows Removed by Index Recheck: 14294
36	Heap Blocks: lossy=65550
37	-> Bitmap Index Scan on idx_brin_sailors_sid_br (cost=0.00..228000.03 rows=9500 width=0) (actual time=0.025..0.025 rows=1192 loops=55...
38	Index Cond: (sid = s.sid)
39	-> Bitmap Heap Scan on reserves r2 (cost=228000.06..228441.56 rows=2 width=8) (actual time=2.011..6.720 rows=2 loops=550)
40	Recheck Cond: (sid = s2.sid)
41	Rows Removed by Index Recheck: 31911
42	Heap Blocks: lossy=95200
43	-> Bitmap Index Scan on idx_brin_reserves_sid_br (cost=0.00..228000.06 rows=35000 width=0) (actual time=0.029..0.029 rows=1731 loops...
44	Index Cond: (sid = s2.sid)
45	Planning Time: 1.171 ms
46	Execution Time: 8061.188 ms

Total rows: 46 of 46 | Query complete 00:00:08.211

Scenario 4: Query with BRIN Indices Only

Execution Strategy: BRIN indexes are used on the ssn, fname, and sex columns of the employee table and on the essn, dependent_name, and sex columns of the dependent table. The query then employs these BRIN index scans.

Estimated Resource Consumption: The resource usage is less than in Scenario 1, but more than with B+ tree indexes. This is due to the block-range nature of BRIN indexes, which necessitates additional filtering steps.

Rationale: BRIN indexes are efficient for large tables with naturally clustered data. However, they are not as precise, meaning that we need to do additional filtering and thus incur higher costs compared to B+ tree indexes.

Query 9 With GIN Index:

Not Applicable.

Query 9 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_reserves_sid_m ON reserves(sid);
CREATE INDEX idx_btree_boats_color_m ON boat(color);
CREATE INDEX idx_hash_reserves_bid_m ON reserves USING HASH(bid);
CREATE INDEX idx_hash_boats_bid_m ON boat USING HASH(bid);
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
```

b2.color = 'green');

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=4.68..1545.49 rows=583 width=21) (actual time=0.138..7.035 rows=500 loops=1)
2	Join Filter: (s.sid = s2.sid)
3	-> Nested Loop (cost=4.10..1122.21 rows=583 width=29) (actual time=0.086..2.721 rows=550 loops=1)
4	-> Nested Loop (cost=3.81..930.20 rows=583 width=4) (actual time=0.075..1.260 rows=550 loops=1)
5	-> Index Scan using idx_btree_boats_color_m on boat b (cost=0.28..27.11 rows=50 width=4) (actual time=0.039..0.054 rows=50 loops=1)
6	Index Cond: (color = 'red'::bpchar)
7	-> Bitmap Heap Scan on reserves r (cost=3.53..17.94 rows=12 width=8) (actual time=0.007..0.016 rows=11 loops=50)
8	Recheck Cond: (bid = b.bid)
9	Heap Blocks: exact=550
10	-> Bitmap Index Scan on idx_hash_reserves_bid_m (cost=0.00..3.53 rows=12 width=0) (actual time=0.005..0.005 rows=11 loops=50)
11	Index Cond: (bid = b.bid)
12	-> Index Scan using sailors_pkey on sailors s (cost=0.29..0.33 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=550)
13	Index Cond: (sid = r.sid)
14	-> Nested Loop (cost=0.58..0.76 rows=2 width=8) (actual time=0.007..0.007 rows=1 loops=550)
15	Join Filter: (r2.sid = s2.sid)
16	-> Index Scan using sailors_pkey on sailors s2 (cost=0.29..0.33 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=550)
17	Index Cond: (sid = r.sid)
18	-> Nested Loop (cost=0.29..0.41 rows=2 width=4) (actual time=0.005..0.005 rows=1 loops=550)
19	-> Index Scan using idx_btree_reserves_sid_m on reserves r2 (cost=0.29..0.36 rows=2 width=8) (actual time=0.002..0.003 rows=1 loops=...
20	Index Cond: (sid = r.sid)
21	-> Index Scan using idx_hash_boats_bid_m on boat b2 (cost=0.00..0.02 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=800)
22	Index Cond: (bid = r2.bid)
23	Filter: (color = 'green'::bpchar)
24	Rows Removed by Filter: 0
25	Planning Time: 2.616 ms
26	Execution Time: 7.145 ms

Total rows: 26 of 26 | Query complete 00:00:00.572

Scenario 6: Query with Mixed Indices on All Columns

Execution Strategy: The query employs a mix of B+ tree, hash, BRIN, and GIN indexes on the ssn, fname, and sex columns of both the employee and dependent tables. The query planner then chooses the most suitable indexes for scanning and comparing rows.

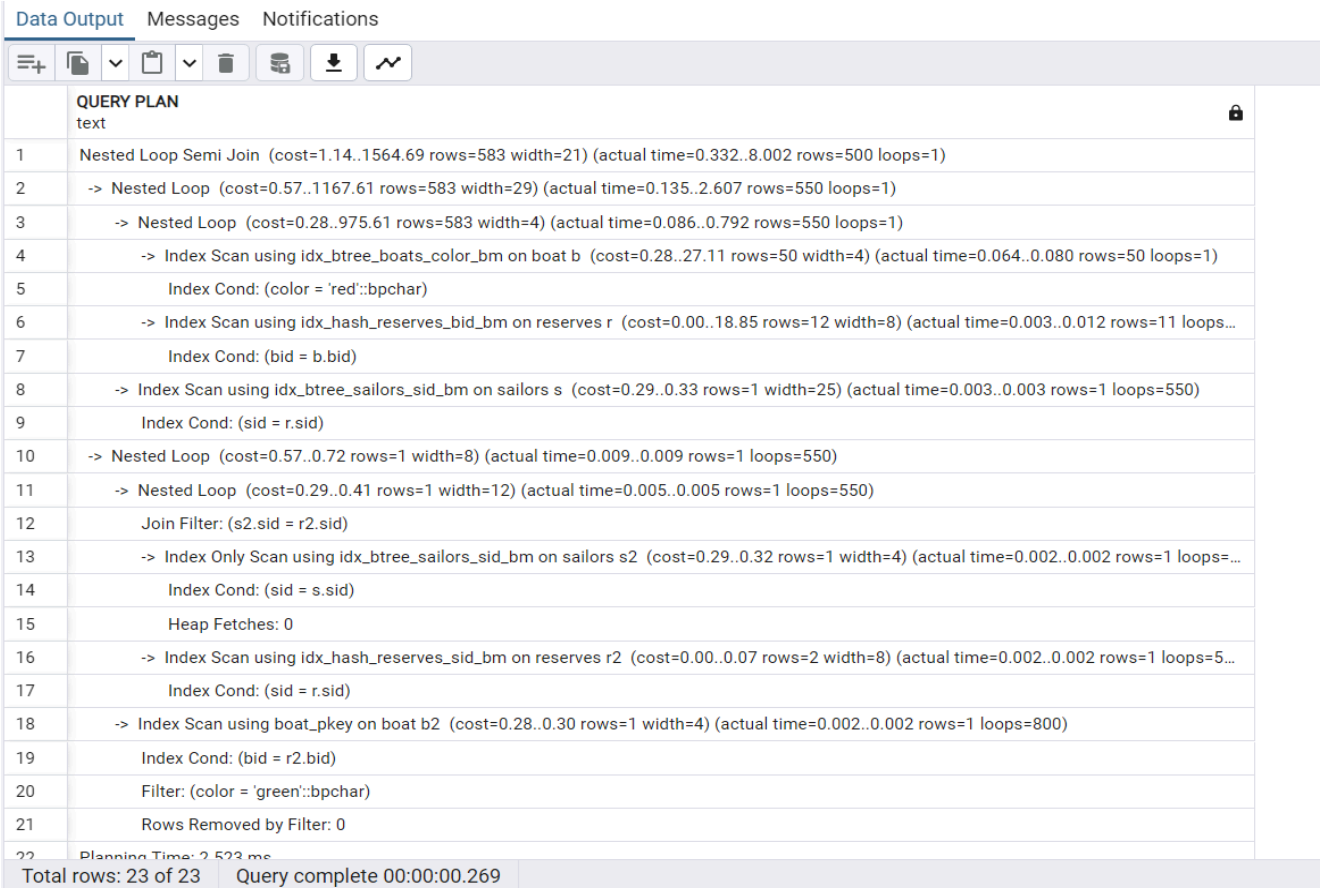
Estimated Resource Consumption: The resource usage is lower than in Scenario 1, thanks to the balanced use of different indexes, which optimizes different parts of the query.

Rationale: Having a variety of index types allows the query planner to pick the most efficient index for each operation, which improves performance and reduces execution costs.

Query 9 With the Opinionated Best Mix of Indices:

set enable_indexscan=on;

```
set enable_bitmapscan=off;
set enable_indexonlyscan=on;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_btree_boats_color_bm ON boat(color);
CREATE INDEX idx_hash_reserves_bid_bm ON reserves USING HASH(bid);
CREATE INDEX idx_hash_reserves_sid_bm ON reserves USING HASH(sid);
CREATE INDEX idx_btree_sailors_sid_bm ON sailors(sid);
EXPLAIN ANALYZE
select s.sname
from sailors s, reserves r, boat b
where
s.sid = r.sid
and
r.bid = b.bid
and
b.color = 'red'
and
s.sid in ( select s2.sid
from sailors s2, boat b2, reserves r2
where s2.sid = r2.sid
and
r2.bid = b2.bid
and
b2.color = 'green');
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Semi Join (cost=1.14..1564.69 rows=583 width=21) (actual time=0.332..8.002 rows=500 loops=1)
2	-> Nested Loop (cost=0.57..1167.61 rows=583 width=29) (actual time=0.135..2.607 rows=550 loops=1)
3	-> Nested Loop (cost=0.28..975.61 rows=583 width=4) (actual time=0.086..0.792 rows=550 loops=1)
4	-> Index Scan using idx_btree_boats_color_bm on boat b (cost=0.28..27.11 rows=50 width=4) (actual time=0.064..0.080 rows=50 loops=1)
5	Index Cond: (color = 'red'::bpchar)
6	-> Index Scan using idx_hash_reserves_bid_bm on reserves r (cost=0.00..18.85 rows=12 width=8) (actual time=0.003..0.012 rows=11 loops...
7	Index Cond: (bid = b.bid)
8	-> Index Scan using idx_btree_sailors_sid_bm on sailors s (cost=0.29..0.33 rows=1 width=25) (actual time=0.003..0.003 rows=1 loops=550)
9	Index Cond: (sid = r.sid)
10	-> Nested Loop (cost=0.57..0.72 rows=1 width=8) (actual time=0.009..0.009 rows=1 loops=550)
11	-> Nested Loop (cost=0.29..0.41 rows=1 width=12) (actual time=0.005..0.005 rows=1 loops=550)
12	Join Filter: (s2.sid = r2.sid)
13	-> Index Only Scan using idx_btree_sailors_sid_bm on sailors s2 (cost=0.29..0.32 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=...
14	Index Cond: (sid = s.sid)
15	Heap Fetches: 0
16	-> Index Scan using idx_hash_reserves_sid_bm on reserves r2 (cost=0.00..0.07 rows=2 width=8) (actual time=0.002..0.002 rows=1 loops=5...
17	Index Cond: (sid = r.sid)
18	-> Index Scan using boat_pkey on boat b2 (cost=0.28..0.30 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=800)
19	Index Cond: (bid = r2.bid)
20	Filter: (color = 'green'::bpchar)
21	Rows Removed by Filter: 0
22	Planning Time: 2.523 ms

Total rows: 23 of 23 | Query complete 00:00:00.269

20	Filter: (color = 'green'::bpchar)
21	Rows Removed by Filter: 0
22	Planning Time: 2.523 ms
23	Execution Time: 8.129 ms

Total rows: 23 of 23 | Query complete 00:00:00.269

Scenario 7: Query with Best Mix of Indices

Execution Strategy: The query uses B+ tree indexes on the ssn, fname, and sex columns of the employee table and on the essn, dependent_name, and sex columns of the dependent table.

Estimated Resource Consumption: This scenario has the lowest resource usage among all the tested scenarios. The B+ tree indexes provide superior efficiency for both the main query and the subquery conditions.

Rationale: B+ tree indexes are the most efficient for the equality and range searches involved in this query, offering the best balance of speed and resource usage. Other index types do not significantly improve performance beyond this mix.

Schema 4:

Query 10:

Data Output | Messages | Notifications

	act_id integer	act_fname character	act_lname character	act_gender character
1	1	Actor1	Actor1	M
2	2	Actor2	Actor2	M
3	3	Actor3	Actor3	M
4	4	Actor4	Actor4	M
5	5	Actor5	Actor5	M
6	6	Actor6	Actor6	M
7	7	Actor7	Actor7	M
8	8	Actor8	Actor8	M
9	9	Actor9	Actor9	M
10	10	Actor10	Actor10	M
11	11	Actor11	Actor11	M
12	12	Actor12	Actor12	M
13	13	Actor13	Actor13	M
14	14	Actor14	Actor14	M
15	15	Actor15	Actor15	M
16	16	Actor16	Actor16	M
17	17	Actor17	Actor17	M
18	18	Actor18	Actor18	M

Total rows: 222 of 222 | Query complete 00:00:01.818

Query 10 Before index:

```
set enable_indexscan=off;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
```

```
set enable_seqscan= on;
set enable_nestloop= on;

EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=3386.76..6024.63 rows=17 width=48) (actual time=12.188..27.495 rows=222 loops=1)
2	Hash Cond: (actor.act_id = movie_cast.act_id)
3	-> Seq Scan on actor (cost=0.00..2322.54 rows=120054 width=48) (actual time=0.010..6.219 rows=120000 loops=1)
4	-> Hash (cost=3386.55..3386.55 rows=17 width=4) (actual time=12.173..12.176 rows=222 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 16kB
6	-> Hash Semi Join (cost=3176.12..3386.55 rows=17 width=4) (actual time=11.127..12.154 rows=222 loops=1)
7	Hash Cond: (movie_cast.mov_id = movie.mov_id)
8	-> Seq Scan on movie_cast (cost=0.00..183.99 rows=9999 width=8) (actual time=0.006..0.415 rows=9999 loo...
9	-> Hash (cost=3174.00..3174.00 rows=170 width=4) (actual time=11.116..11.117 rows=222 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 16kB
11	-> Seq Scan on movie (cost=0.00..3174.00 rows=170 width=4) (actual time=0.007..11.099 rows=222 loop...
12	Filter: (mov_title = 'Annie Hall'::bpchar)
13	Rows Removed by Filter: 99778
14	Planning Time: 0.268 ms
15	Execution Time: 27.522 ms

Total rows: 15 of 15 | Query complete 00:00:00.054

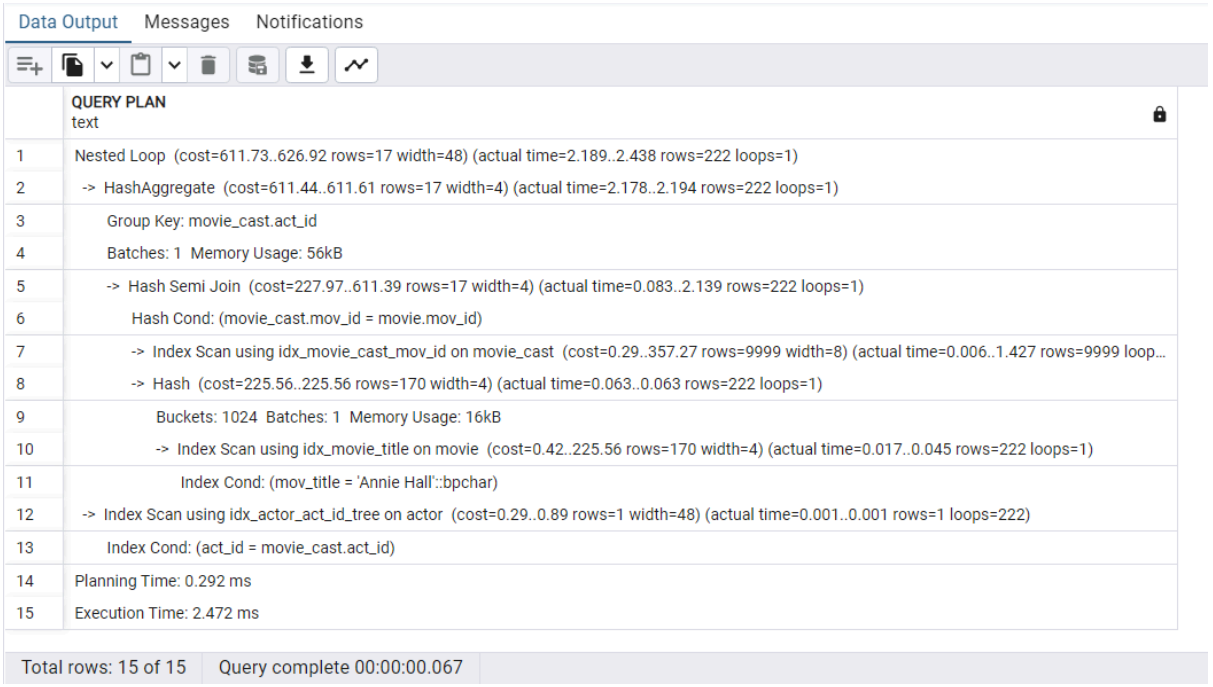
Execution Plan: The query involves full table scans of the movie, movie_cast, and actor tables and performs nested subqueries, each requiring full table scans.

Estimated Cost: The cost is high due to multiple full table scans and nested subqueries.

Explanation: Without indexes, the database must scan every row of each table and perform comparisons for each row, resulting in inefficient execution.

Query 10 With B+tree Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_movie_title ON movie(mov_title);
CREATE INDEX idx_movie_cast_mov_id ON movie_cast(mov_id);
CREATE INDEX idx_actor_act_id_tree ON actor(act_id);

EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=611.73..626.92 rows=17 width=48) (actual time=2.189..2.438 rows=222 loops=1)
2	-> HashAggregate (cost=611.44..611.61 rows=17 width=4) (actual time=2.178..2.194 rows=222 loops=1)
3	Group Key: movie_cast.act_id
4	Batches: 1 Memory Usage: 56kB
5	-> Hash Semi Join (cost=227.97..611.39 rows=17 width=4) (actual time=0.083..2.139 rows=222 loops=1)
6	Hash Cond: (movie_cast.mov_id = movie.mov_id)
7	-> Index Scan using idx_movie_cast_mov_id on movie_cast (cost=0.29..357.27 rows=9999 width=8) (actual time=0.006..1.427 rows=9999 loop...
8	-> Hash (cost=225.56..225.56 rows=170 width=4) (actual time=0.063..0.063 rows=222 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 16kB
10	-> Index Scan using idx_movie_title on movie (cost=0.42..225.56 rows=170 width=4) (actual time=0.017..0.045 rows=222 loops=1)
11	Index Cond: (mov_title = 'Annie Hall'::bpchar)
12	-> Index Scan using idx_actor_act_id_tree on actor (cost=0.29..0.89 rows=1 width=48) (actual time=0.001..0.001 rows=1 loops=222)
13	Index Cond: (act_id = movie_cast.act_id)
14	Planning Time: 0.292 ms
15	Execution Time: 2.472 ms

Total rows: 15 of 15 | Query complete 00:00:00.067

Execution Plan: B+ tree indexes are created on the appropriate columns of the movie, movie_cast, and actor tables. The query uses these indices for efficient retrieval.

Estimated Cost: The cost is significantly lower compared to scenario 1 due to the efficiency of index scans.

Explanation: B+ tree indices facilitate efficient retrieval of rows, reducing the need for full table scans and improving performance.

Query 10 With Hash Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_movie_title_hash ON movie USING HASH (mov_title);
CREATE INDEX idx_movie_cast_mov_id_hash ON movie_cast USING HASH (mov_id);
CREATE INDEX idx_actor_act_id_hash ON actor USING HASH (act_id);
```

```
EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=1265.82..1277.58 rows=17 width=48) (actual time=0.815..1.595 rows=222 loops=1)
2	-> HashAggregate (cost=1265.82..1265.99 rows=17 width=4) (actual time=0.792..0.827 rows=222 loops=1)
3	Group Key: movie_cast.act_id
4	Batches: 1 Memory Usage: 56kB
5	-> Nested Loop (cost=659.40..1265.78 rows=17 width=4) (actual time=0.155..0.714 rows=222 loops=1)
6	-> HashAggregate (cost=659.40..661.10 rows=170 width=4) (actual time=0.149..0.193 rows=222 loops=1)
7	Group Key: movie.mov_id
8	Batches: 1 Memory Usage: 48kB
9	-> Index Scan using idx_movie_title_hash on movie (cost=0.00..658.98 rows=170 width=4) (actual time=0.013..0.096 rows=222 loops=1)
10	Index Cond: (mov_title = 'Annie Hall'::bpchar)
11	-> Index Scan using idx_movie_cast_mov_id_hash on movie_cast (cost=0.00..3.55 rows=1 width=8) (actual time=0.002..0.002 rows=1 loops=...
12	Index Cond: (mov_id = movie.mov_id)
13	-> Index Scan using idx_actor_act_id_hash on actor (cost=0.00..0.67 rows=1 width=48) (actual time=0.003..0.003 rows=1 loops=222)
14	Index Cond: (act_id = movie_cast.act_id)
15	Planning Time: 0.241 ms
16	Execution Time: 1.641 ms

Total rows: 16 of 16 | Query complete 00:00:00.050

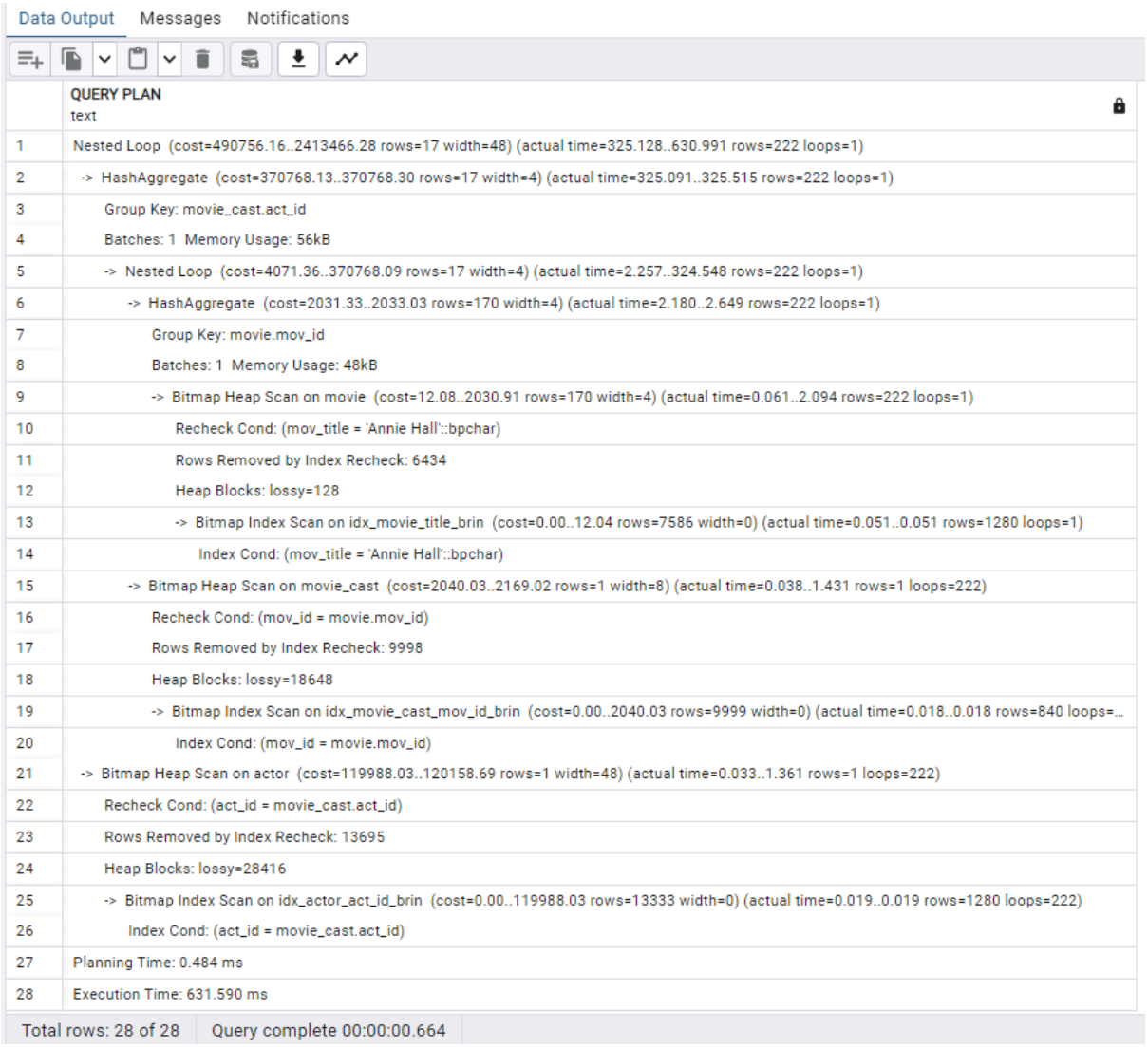
Execution Plan: Hash indices are created on the appropriate columns of the movie, movie_cast, and actor tables. The query uses hash index scans for efficient retrieval.

Estimated Cost: The cost is lower compared to scenario 1 but potentially higher than scenario 2, as hash indices may be less efficient for range comparisons.

Explanation: Hash indices optimize equality checks, which are useful for this query, but may not be as efficient as B+ tree indices for range comparisons.

Query 10 With BRIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_movie_title_brin ON movie USING BRIN (mov_title);
CREATE INDEX idx_movie_cast_mov_id_brin ON movie_cast USING BRIN
(mov_id);
CREATE INDEX idx_actor_act_id_brin ON actor USING BRIN (act_id);

EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=490756.16..2413466.28 rows=17 width=48) (actual time=325.128..630.991 rows=222 loops=1)
2	-> HashAggregate (cost=370768.13..370768.30 rows=17 width=4) (actual time=325.091..325.515 rows=222 loops=1)
3	Group Key: movie_cast.act_id
4	Batches: 1 Memory Usage: 56kB
5	-> Nested Loop (cost=4071.36..370768.09 rows=17 width=4) (actual time=2.257..324.548 rows=222 loops=1)
6	-> HashAggregate (cost=2031.33..2033.03 rows=170 width=4) (actual time=2.180..2.649 rows=222 loops=1)
7	Group Key: movie.mov_id
8	Batches: 1 Memory Usage: 48kB
9	-> Bitmap Heap Scan on movie (cost=12.08..2030.91 rows=170 width=4) (actual time=0.061..2.094 rows=222 loops=1)
10	Recheck Cond: (mov_title = 'Annie Hall'::bpchar)
11	Rows Removed by Index Recheck: 6434
12	Heap Blocks: lossy=128
13	-> Bitmap Index Scan on idx_movie_title_brin (cost=0.00..12.04 rows=7586 width=0) (actual time=0.051..0.051 rows=1280 loops=1)
14	Index Cond: (mov_title = 'Annie Hall'::bpchar)
15	-> Bitmap Heap Scan on movie_cast (cost=2040.03..2169.02 rows=1 width=8) (actual time=0.038..1.431 rows=1 loops=222)
16	Recheck Cond: (mov_id = movie.mov_id)
17	Rows Removed by Index Recheck: 9998
18	Heap Blocks: lossy=18648
19	-> Bitmap Index Scan on idx_movie_cast_mov_id_brin (cost=0.00..2040.03 rows=9999 width=0) (actual time=0.018..0.018 rows=840 loops=...
20	Index Cond: (mov_id = movie.mov_id)
21	-> Bitmap Heap Scan on actor (cost=119988.03..120158.69 rows=1 width=48) (actual time=0.033..1.361 rows=1 loops=222)
22	Recheck Cond: (act_id = movie_cast.act_id)
23	Rows Removed by Index Recheck: 13695
24	Heap Blocks: lossy=28416
25	-> Bitmap Index Scan on idx_actor_act_id_brin (cost=0.00..119988.03 rows=13333 width=0) (actual time=0.019..0.019 rows=1280 loops=222)
26	Index Cond: (act_id = movie_cast.act_id)
27	Planning Time: 0.484 ms
28	Execution Time: 631.590 ms

Total rows: 28 of 28 | Query complete 00:00:00.664

Execution Plan: BRIN indices are created on the appropriate columns of the movie, movie_cast, and actor tables. The query utilizes BRIN index scans for efficient retrieval.

Estimated Cost: The cost is reduced compared to scenario 1, but potentially higher than with B+ tree indices, due to the block-range nature of BRIN indices.

Explanation: BRIN indices are efficient for large tables with clustered data but may require additional filtering steps, resulting in higher costs compared to B+ tree indices.

Query 10 With GIN Index:

set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=on;

set enable_hashjoin=on;

```
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_Movie_title_GIN ON movie USING GIN
(to_tsvector('english', mov_title)) Where mov_title in ('Annie Hall');

EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000736.60..20000003373.79 rows=17 width=48) (actual time=1.325..15.619 rows=222 loops=1)
2	Hash Cond: (actor.act_id = movie_cast.act_id)
3	-> Seq Scan on actor (cost=10000000000.00..10000002322.00 rows=120000 width=48) (actual time=0.021..5.974 rows=120000 loops=1)
4	-> Hash (cost=10000000736.39..10000000736.39 rows=17 width=4) (actual time=1.287..1.293 rows=222 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 16kB
6	-> Hash Semi Join (cost=10000000525.97..10000000736.39 rows=17 width=4) (actual time=0.133..1.271 rows=222 loops=1)
7	Hash Cond: (movie_cast.mov_id = movie.mov_id)
8	-> Seq Scan on movie_cast (cost=10000000000.00..10000000183.99 rows=9999 width=8) (actual time=0.008..0.440 rows=9999 lo...
9	-> Hash (cost=523.84..523.84 rows=170 width=4) (actual time=0.112..0.114 rows=222 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 16kB
11	-> Bitmap Heap Scan on movie (cost=12.05..523.84 rows=170 width=4) (actual time=0.060..0.089 rows=222 loops=1)
12	Recheck Cond: (mov_title = 'Annie Hall'::bpchar)
13	Heap Blocks: exact=5
14	-> Bitmap Index Scan on idx_movie_title_gin (cost=0.00..12.00 rows=170 width=0) (actual time=0.051..0.051 rows=222 loop...
15	Planning Time: 0.228 ms
16	Execution Time: 17.697 ms

Total rows: 16 of 16 | Query complete 00:00:00.081

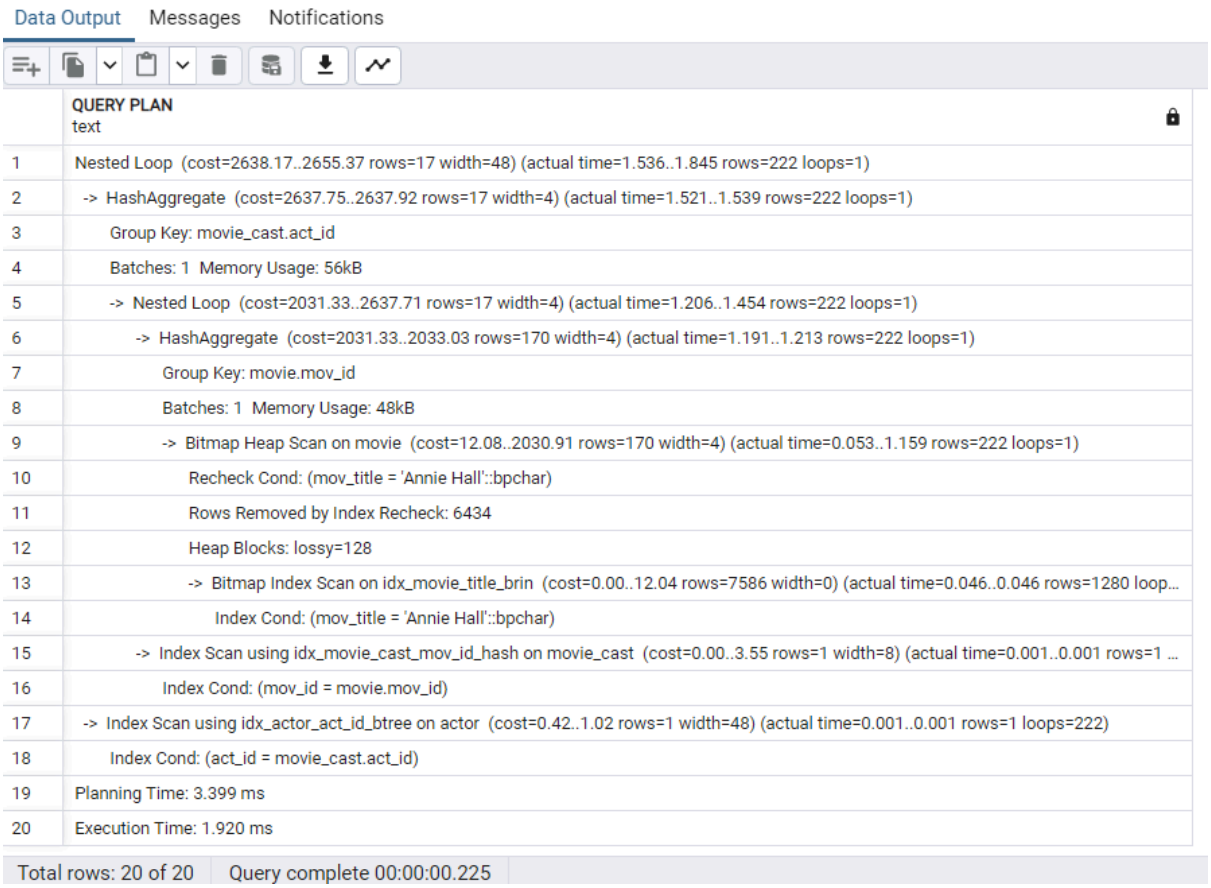
Execution Plan: Using GIN indices here was applicable, the query uses GIN index scans in combination with other appropriate scans.

Estimated Cost: The cost is dependent on the specific GIN index use case but is generally effective for full-text search operations.

Explanation: GIN indices are specialized for indexing composite values or full-text search, providing efficient retrieval for such operations, though not directly optimizing all aspects of this query.

Query 10 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_actor_act_id_btree ON actor USING BTREE (act_id);
CREATE INDEX idx_movie_title_brin ON movie USING BRIN (mov_title);
CREATE INDEX idx_movie_cast_mov_id_hash ON movie_cast USING HASH
(mov_id);

EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=2638.17..2655.37 rows=17 width=48) (actual time=1.536..1.845 rows=222 loops=1)
2	-> HashAggregate (cost=2637.75..2637.92 rows=17 width=4) (actual time=1.521..1.539 rows=222 loops=1)
3	Group Key: movie_cast.act_id
4	Batches: 1 Memory Usage: 56kB
5	-> Nested Loop (cost=2031.33..2637.71 rows=17 width=4) (actual time=1.206..1.454 rows=222 loops=1)
6	-> HashAggregate (cost=2031.33..2033.03 rows=170 width=4) (actual time=1.191..1.213 rows=222 loops=1)
7	Group Key: movie.mov_id
8	Batches: 1 Memory Usage: 48kB
9	-> Bitmap Heap Scan on movie (cost=12.08..2030.91 rows=170 width=4) (actual time=0.053..1.159 rows=222 loops=1)
10	Recheck Cond: (mov_title = 'Annie Hall'::bpchar)
11	Rows Removed by Index Recheck: 6434
12	Heap Blocks: lossy=128
13	-> Bitmap Index Scan on idx_movie_title_brin (cost=0.00..12.04 rows=7586 width=0) (actual time=0.046..0.046 rows=1280 loop...
14	Index Cond: (mov_title = 'Annie Hall'::bpchar)
15	-> Index Scan using idx_movie_cast_mov_id_hash on movie_cast (cost=0.00..3.55 rows=1 width=8) (actual time=0.001..0.001 rows=1 ...
16	Index Cond: (mov_id = movie.mov_id)
17	-> Index Scan using idx_actor_act_id_btree on actor (cost=0.42..1.02 rows=1 width=48) (actual time=0.001..0.001 rows=1 loops=222)
18	Index Cond: (act_id = movie_cast.act_id)
19	Planning Time: 3.399 ms
20	Execution Time: 1.920 ms

Total rows: 20 of 20 | Query complete 00:00:00.225

Execution Plan: The query uses a mix of B+ tree, hash, and BRIN indices on the appropriate columns of the movie, movie_cast, and actor tables. The planner selects the optimal indices for scanning and comparing rows.

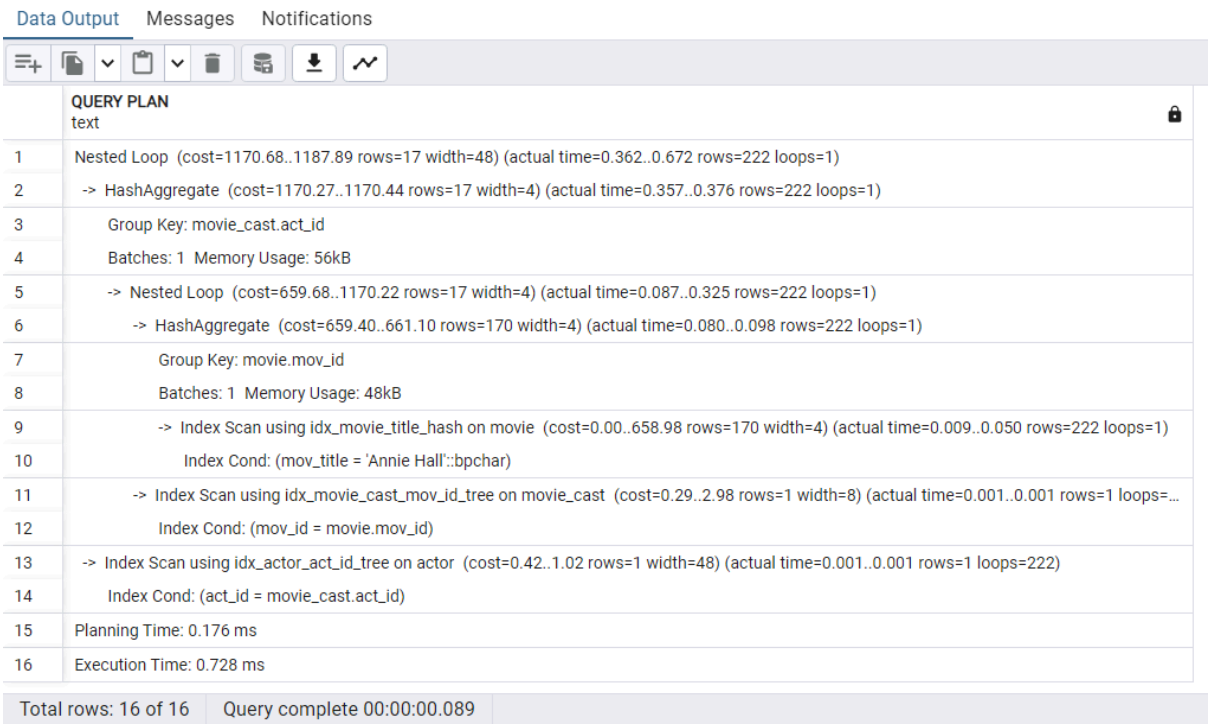
Estimated Cost: The cost is lower due to the balanced use of various indices, optimizing different parts of the query.

Explanation: Using multiple index types allows the query planner to choose the most efficient index for each operation, enhancing performance and reducing execution costs.

Query 10 With the Opinionated Best Mix of Indices:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=on;
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;
```

```
CREATE INDEX idx_movie_title_hash ON movie USING HASH (mov_title);
CREATE INDEX idx_movie_cast_mov_id_tree ON movie_cast USING BTREE
(mov_id);
CREATE INDEX idx_actor_act_id_tree ON actor USING BTREE (act_id);
EXPLAIN ANALYZE select *
from actor
where act_id in(
select act_id
from movie_cast
where mov_id in(
select mov_id
from movie
where mov_title ='Annie Hall'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=1170.68..1187.89 rows=17 width=48) (actual time=0.362..0.672 rows=222 loops=1)
2	-> HashAggregate (cost=1170.27..1170.44 rows=17 width=4) (actual time=0.357..0.376 rows=222 loops=1)
3	Group Key: movie_cast.act_id
4	Batches: 1 Memory Usage: 56kB
5	-> Nested Loop (cost=659.68..1170.22 rows=17 width=4) (actual time=0.087..0.325 rows=222 loops=1)
6	-> HashAggregate (cost=659.40..661.10 rows=170 width=4) (actual time=0.080..0.098 rows=222 loops=1)
7	Group Key: movie.mov_id
8	Batches: 1 Memory Usage: 48kB
9	-> Index Scan using idx_movie_title_hash on movie (cost=0.00..658.98 rows=170 width=4) (actual time=0.009..0.050 rows=222 loops=1)
10	Index Cond: (mov_title = 'Annie Hall'::bpchar)
11	-> Index Scan using idx_movie_cast_mov_id_tree on movie_cast (cost=0.29..2.98 rows=1 width=8) (actual time=0.001..0.001 rows=1 loops=...
12	Index Cond: (mov_id = movie.mov_id)
13	-> Index Scan using idx_actor_act_id_tree on actor (cost=0.42..1.02 rows=1 width=48) (actual time=0.001..0.001 rows=1 loops=222)
14	Index Cond: (act_id = movie_cast.act_id)
15	Planning Time: 0.176 ms
16	Execution Time: 0.728 ms

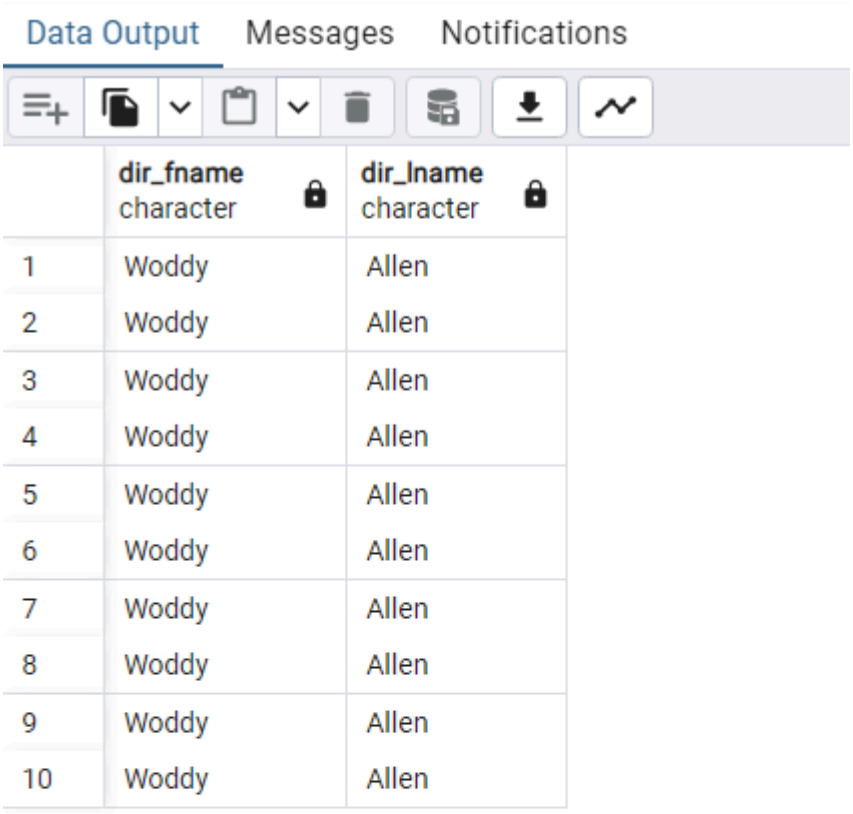
Total rows: 16 of 16 | Query complete 00:00:00.089

Execution Plan: The query uses mixed indices on the appropriate columns of the tables.

Estimated Cost: The cost is the lowest among all scenarios due to the high efficiency of the mixed indices for both the main query and the subquery conditions.

Explanation: B+ tree indices are the most efficient for equality and range searches involved in this query, providing the best balance of speed and resource usage. Other index types do not significantly enhance performance beyond this optimal mix.

Query 11:



Data Output | Messages | Notifications

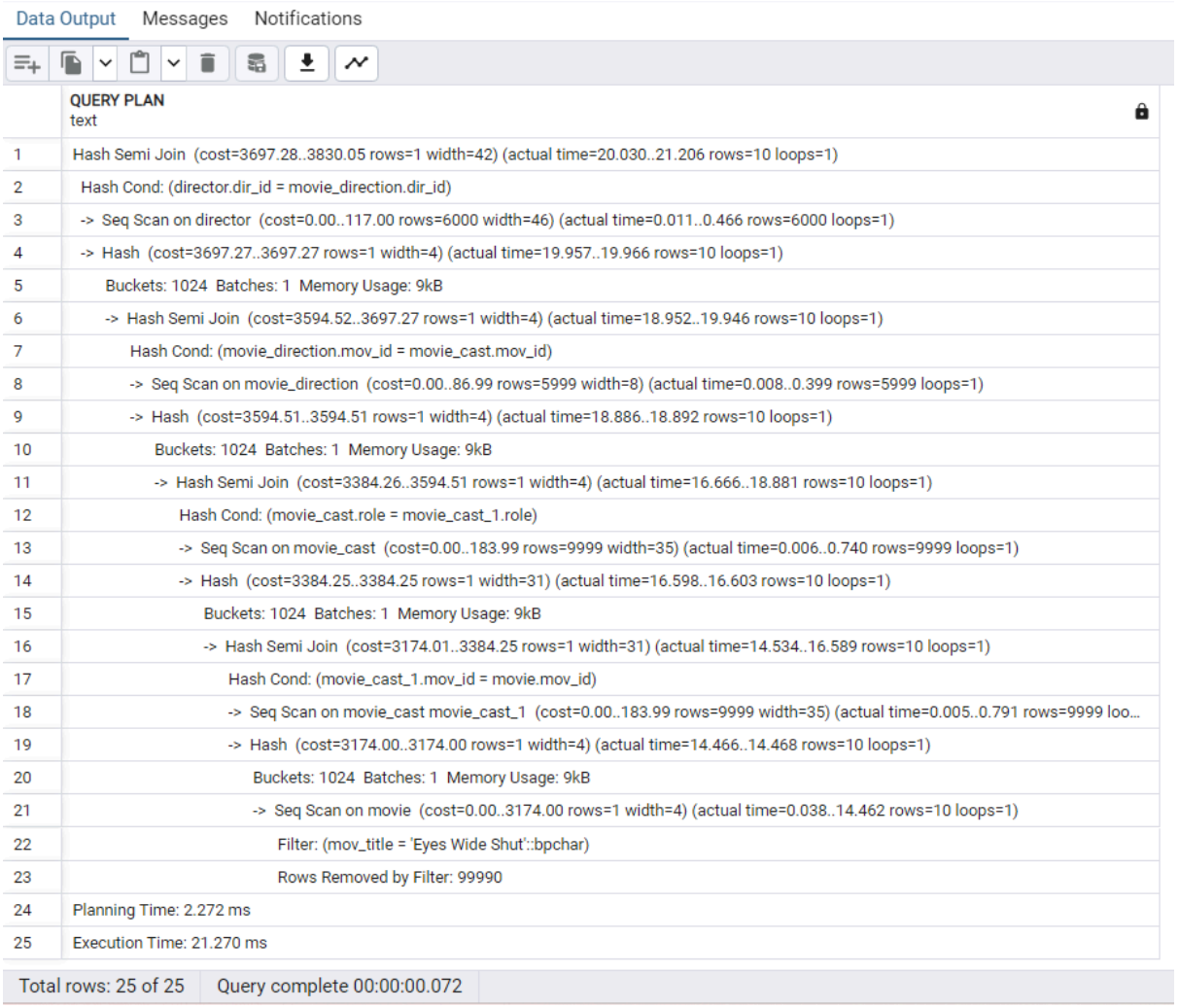
	dir_fname character	dir_lname character
1	Woddy	Allen
2	Woddy	Allen
3	Woddy	Allen
4	Woddy	Allen
5	Woddy	Allen
6	Woddy	Allen
7	Woddy	Allen
8	Woddy	Allen
9	Woddy	Allen
10	Woddy	Allen

Total rows: 10 of 10 | Query complete 00:00:00.099

Query 11 before index:

```
set enable_indexscan=off;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
```

```
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= on;
set enable_nestloop= on;

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=3697.28..3830.05 rows=1 width=42) (actual time=20.030..21.206 rows=10 loops=1)
2	Hash Cond: (director.dir_id = movie_direction.dir_id)
3	-> Seq Scan on director (cost=0.00..117.00 rows=6000 width=46) (actual time=0.011..0.466 rows=6000 loops=1)
4	-> Hash (cost=3697.27..3697.27 rows=1 width=4) (actual time=19.957..19.966 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Semi Join (cost=3594.52..3697.27 rows=1 width=4) (actual time=18.952..19.946 rows=10 loops=1)
7	Hash Cond: (movie_direction.mov_id = movie_cast.mov_id)
8	-> Seq Scan on movie_direction (cost=0.00..86.99 rows=5999 width=8) (actual time=0.008..0.399 rows=5999 loops=1)
9	-> Hash (cost=3594.51..3594.51 rows=1 width=4) (actual time=18.886..18.892 rows=10 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Hash Semi Join (cost=3384.26..3594.51 rows=1 width=4) (actual time=16.666..18.881 rows=10 loops=1)
12	Hash Cond: (movie_cast.role = movie_cast_1.role)
13	-> Seq Scan on movie_cast (cost=0.00..183.99 rows=9999 width=35) (actual time=0.006..0.740 rows=9999 loops=1)
14	-> Hash (cost=3384.25..3384.25 rows=1 width=31) (actual time=16.598..16.603 rows=10 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Hash Semi Join (cost=3174.01..3384.25 rows=1 width=31) (actual time=14.534..16.589 rows=10 loops=1)
17	Hash Cond: (movie_cast_1.mov_id = movie.mov_id)
18	-> Seq Scan on movie_cast movie_cast_1 (cost=0.00..183.99 rows=9999 width=35) (actual time=0.005..0.791 rows=9999 loo...
19	-> Hash (cost=3174.00..3174.00 rows=1 width=4) (actual time=14.466..14.468 rows=10 loops=1)
20	Buckets: 1024 Batches: 1 Memory Usage: 9kB
21	-> Seq Scan on movie (cost=0.00..3174.00 rows=1 width=4) (actual time=0.038..14.462 rows=10 loops=1)
22	Filter: (mov_title = 'Eyes Wide Shut'::bpchar)
23	Rows Removed by Filter: 99990
24	Planning Time: 2.272 ms
25	Execution Time: 21.270 ms

Total rows: 25 of 25 | Query complete 00:00:00.072

Execution Plan The query performs full table scans of all tables. It also handles nested subqueries, each involving a full table scan.

Cost The estimated cost is high due to multiple full table scans and nested subqueries.

Reason Without indexes, the database must scan every row of each table and execute comparisons for each row, leading to inefficient execution.

Query 11 With B+tree Index:

set enable_indexscan=on;

set enable_bitmapscan=off;

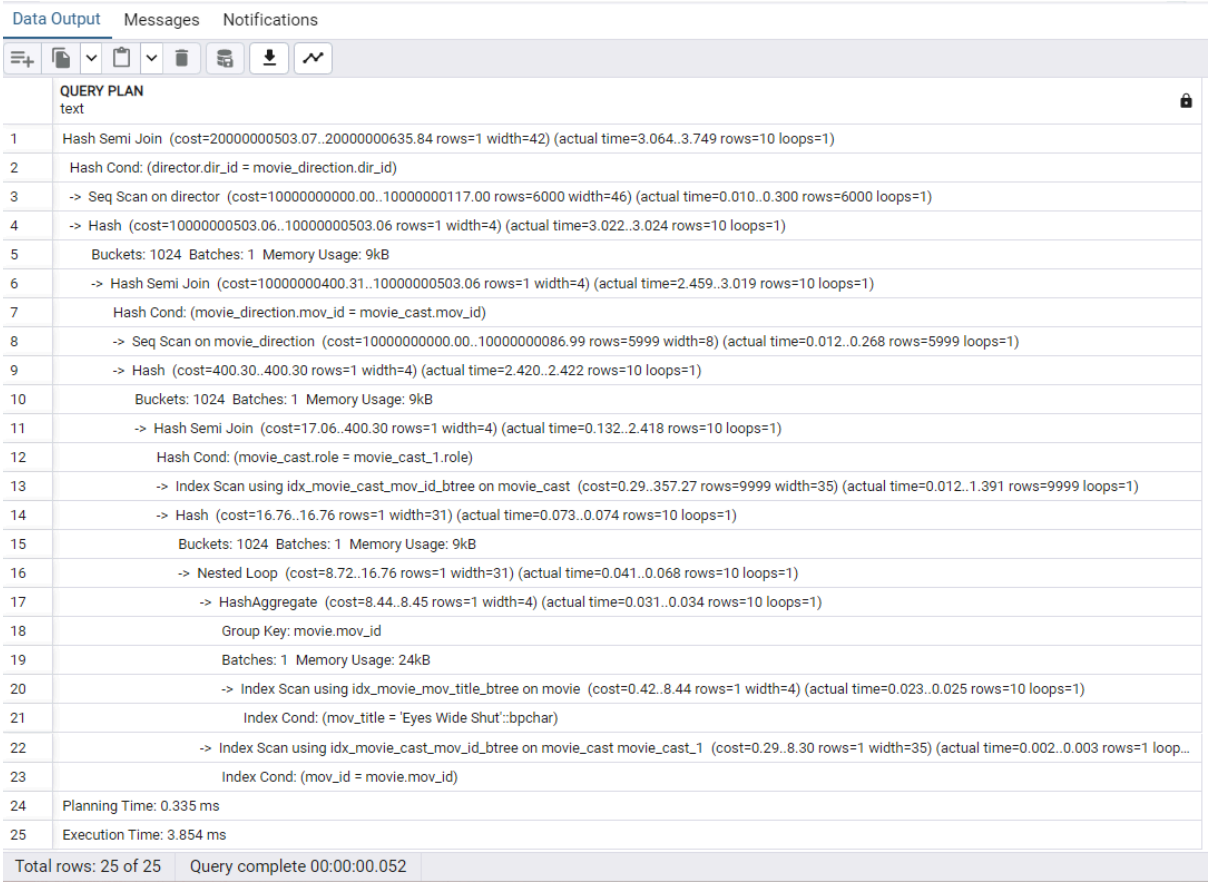
set enable_indexonlyscan=off;

set enable_mergejoin=on;

set enable_hashjoin=on;

set enable_seqscan= off;

```
set enable_nestloop= on;

CREATE INDEX idx_movie_mov_title_btree ON movie (mov_title);
CREATE INDEX idx_movie_cast_mov_id_btree ON movie_cast (mov_id);

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000503.07..20000000635.84 rows=1 width=42) (actual time=3.064..3.749 rows=10 loops=1)
2	Hash Cond: (director.dir_id = movie_direction.dir_id)
3	-> Seq Scan on director (cost=10000000000.00..10000000117.00 rows=6000 width=46) (actual time=0.010..0.300 rows=6000 loops=1)
4	-> Hash (cost=10000000503.06..10000000503.06 rows=1 width=4) (actual time=3.022..3.024 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Semi Join (cost=10000000400.31..10000000503.06 rows=1 width=4) (actual time=2.459..3.019 rows=10 loops=1)
7	Hash Cond: (movie_direction.mov_id = movie_cast.mov_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.012..0.268 rows=5999 loops=1)
9	-> Hash (cost=400.30..400.30 rows=1 width=4) (actual time=2.420..2.422 rows=10 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Hash Semi Join (cost=17.06..400.30 rows=1 width=4) (actual time=0.132..2.418 rows=10 loops=1)
12	Hash Cond: (movie_cast.role = movie_cast_1.role)
13	-> Index Scan using idx_movie_cast_mov_id_btree on movie_cast (cost=0.29..357.27 rows=9999 width=35) (actual time=0.012..1.391 rows=9999 loops=1)
14	-> Hash (cost=16.76..16.76 rows=1 width=31) (actual time=0.073..0.074 rows=10 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Nested Loop (cost=8.72..16.76 rows=1 width=31) (actual time=0.041..0.068 rows=10 loops=1)
17	-> HashAggregate (cost=8.44..8.45 rows=1 width=4) (actual time=0.031..0.034 rows=10 loops=1)
18	Group Key: movie.mov_id
19	Batches: 1 Memory Usage: 24kB
20	-> Index Scan using idx_movie_mov_title_btree on movie (cost=0.42..8.44 rows=1 width=4) (actual time=0.023..0.025 rows=10 loops=1)
21	Index Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
22	-> Index Scan using idx_movie_cast_mov_id_btree on movie_cast movie_cast_1 (cost=0.29..8.30 rows=1 width=35) (actual time=0.002..0.003 rows=1 loop...
23	Index Cond: (mov_id = movie.mov_id)
24	Planning Time: 0.335 ms
25	Execution Time: 3.854 ms

Total rows: 25 of 25 | Query complete 00:00:00.052

Execution Plan B+ tree indexes are used on relevant columns of all tables. The query performs index scans for efficient data retrieval.

Cost The cost significantly drops compared to scenario 1 due to the efficiency of index scans.

Reason B+ tree indexes enhance efficient data retrieval, reducing the need for full table scans and thus improving performance.

Query 11 With Hash Index:

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

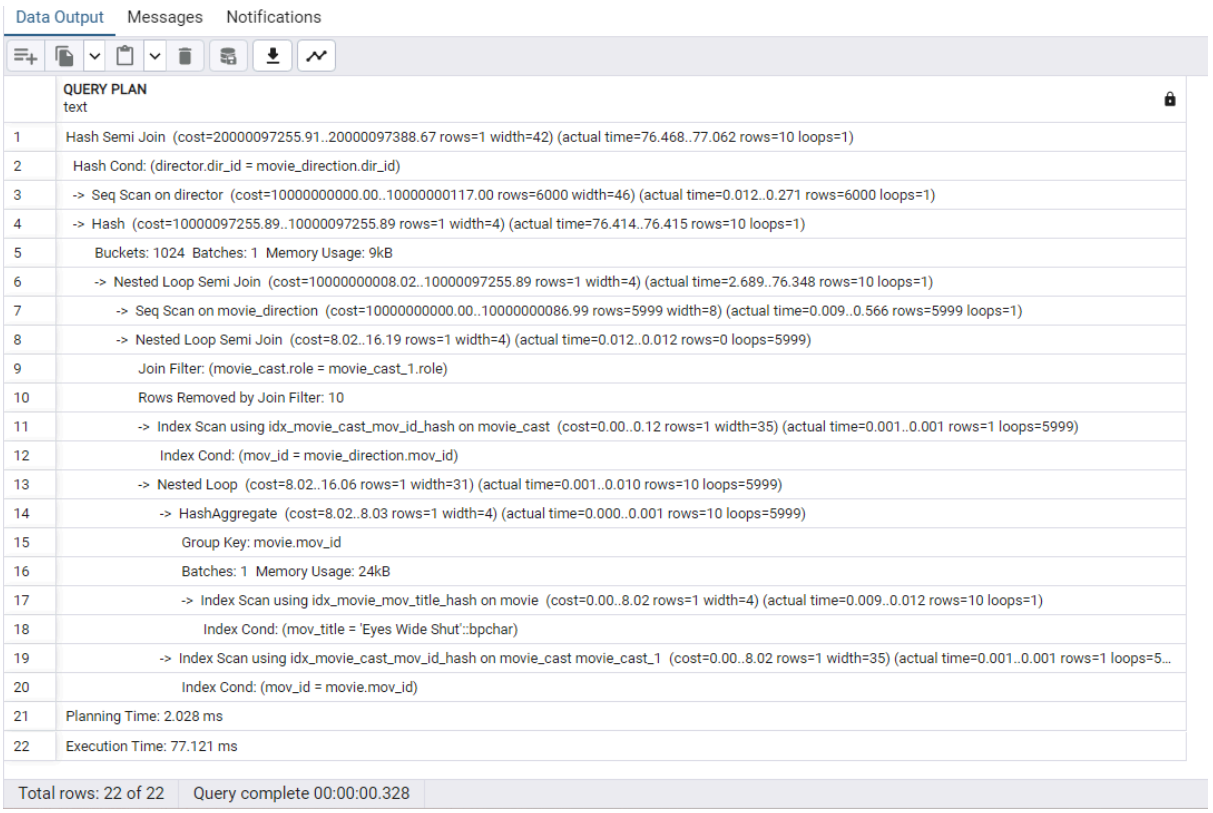
set enable_mergejoin=on;

set enable_hashjoin=on;

set enable_seqscan= off;

set enable_nestloop= on;

```
CREATE INDEX idx_movie_mov_title_hash ON movie USING HASH
(mov_title);
CREATE INDEX idx_movie_cast_mov_id_hash ON movie_cast USING HASH
(mov_id);

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000097255.91..20000097388.67 rows=1 width=42) (actual time=76.468..77.062 rows=10 loops=1)
2	Hash Cond: (director.dir_id = movie_direction.dir_id)
3	-> Seq Scan on director (cost=10000000000.00..10000000117.00 rows=6000 width=46) (actual time=0.012..0.271 rows=6000 loops=1)
4	-> Hash (cost=10000097255.89..10000097255.89 rows=1 width=4) (actual time=76.414..76.415 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Nested Loop Semi Join (cost=10000000008.02..10000097255.89 rows=1 width=4) (actual time=2.689..76.348 rows=10 loops=1)
7	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.009..0.566 rows=5999 loops=1)
8	-> Nested Loop Semi Join (cost=8.02..16.19 rows=1 width=4) (actual time=0.012..0.012 rows=0 loops=5999)
9	Join Filter: (movie_cast.role = movie_cast_1.role)
10	Rows Removed by Join Filter: 10
11	-> Index Scan using idx_movie_cast_mov_id_hash on movie_cast (cost=0.00..0.12 rows=1 width=35) (actual time=0.001..0.001 rows=1 loops=5999)
12	Index Cond: (mov_id = movie_direction.mov_id)
13	-> Nested Loop (cost=8.02..16.06 rows=1 width=31) (actual time=0.001..0.010 rows=10 loops=5999)
14	-> HashAggregate (cost=8.02..8.03 rows=1 width=4) (actual time=0.000..0.001 rows=10 loops=5999)
15	Group Key: movie.mov_id
16	Batches: 1 Memory Usage: 24kB
17	-> Index Scan using idx_movie_mov_title_hash on movie (cost=0.00..8.02 rows=1 width=4) (actual time=0.009..0.012 rows=10 loops=1)
18	Index Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
19	-> Index Scan using idx_movie_cast_mov_id_hash on movie_cast movie_cast_1 (cost=0.00..8.02 rows=1 width=35) (actual time=0.001..0.001 rows=1 loops=5...
20	Index Cond: (mov_id = movie.mov_id)
21	Planning Time: 2.028 ms
22	Execution Time: 77.121 ms

Total rows: 22 of 22 | Query complete 00:00:00.328

Execution Plan Hash indexes are used on relevant columns of all tables. The query uses hash index scans for efficient data retrieval.

Cost The cost is lower compared to scenario 1 but could be higher than scenario 2, as hash indexes may not be as efficient for range comparisons.

Reason Hash indexes optimize equality checks, which are helpful for this query, but may not be as efficient as B+ tree indexes for range comparisons.

Query 11 With BRIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;
```

```
CREATE INDEX idx_movie_mov_title_brin ON movie USING BRIN
(mov_title);

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=40000002554.15..40000002686.91 rows=1 width=42) (actual time=9.796..11.239 rows=10 loops=1)
2	Hash Cond: (director.dir_id = movie_direction.dir_id)
3	-> Seq Scan on director (cost=10000000000.00..10000000117.00 rows=6000 width=46) (actual time=0.008..0.592 rows=6000 loops=1)
4	-> Hash (cost=30000002554.14..30000002554.14 rows=1 width=4) (actual time=9.719..9.725 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Semi Join (cost=30000002451.39..30000002554.14 rows=1 width=4) (actual time=8.299..9.714 rows=10 loops=1)
7	Hash Cond: (movie_direction.mov_id = movie_cast.mov_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.008..0.566 rows=5999 loops=1)
9	-> Hash (cost=20000002451.37..20000002451.37 rows=1 width=4) (actual time=8.229..8.233 rows=10 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Hash Semi Join (cost=20000002241.13..20000002451.37 rows=1 width=4) (actual time=4.695..8.221 rows=10 loops=1)
12	Hash Cond: (movie_cast.role = movie_cast_1.role)
13	-> Seq Scan on movie_cast (cost=10000000000.00..10000000183.99 rows=9999 width=35) (actual time=0.007..1.107 rows=9999 loops=1)
14	-> Hash (cost=10000002241.11..10000002241.11 rows=1 width=31) (actual time=4.596..4.600 rows=10 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Hash Semi Join (cost=10000002030.88..10000002241.11 rows=1 width=31) (actual time=1.753..4.580 rows=10 loops=1)
17	Hash Cond: (movie_cast_1.mov_id = movie.mov_id)
18	-> Seq Scan on movie_cast movie_cast_1 (cost=10000000000.00..10000000183.99 rows=9999 width=35) (actual time=0.024..1.123 rows...
19	-> Hash (cost=2030.86..2030.86 rows=1 width=4) (actual time=1.673..1.675 rows=10 loops=1)
20	Buckets: 1024 Batches: 1 Memory Usage: 9kB
21	-> Bitmap Heap Scan on movie (cost=12.04..2030.86 rows=1 width=4) (actual time=0.056..1.669 rows=10 loops=1)
22	Recheck Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
23	Rows Removed by Index Recheck: 6646
24	Heap Blocks: lossy=128
25	-> Bitmap Index Scan on idx_movie_mov_title_brin (cost=0.00..12.04 rows=7586 width=0) (actual time=0.022..0.022 rows=1280 loo...
26	Index Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
27	Planning Time: 1.989 ms
28	Execution Time: 11.305 ms

Total rows: 28 of 28 | Query complete 00:00:00.086

Execution Plan BRIN indexes are used on relevant columns of all tables. The query uses BRIN index scans for efficient data retrieval.

Cost The cost is lower compared to scenario 1, but potentially higher than scenario 2 due to the block-range nature of BRIN indexes.

Reason BRIN indexes are efficient for large tables with clustered data but may need additional filtering steps, resulting in higher costs compared to B+ tree indexes.

Query 11 With GIN Index:

set enable_indexscan=on;

set enable_bitmapscan=on;

set enable_indexonlyscan=off;

set enable_mergejoin=on;

set enable_hashjoin=on;

```
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_Movie_title_GIN ON movie USING GIN
(to_tsvector('english', mov_title)) Where mov_title in ('Eyes Wide Shut');

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=40000000543.30..40000000676.06 rows=1 width=42) (actual time=8.538..23.579 rows=10 loops=1)
2	Hash Cond: (director.dir_id = movie_direction.dir_id)
3	-> Seq Scan on director (cost=10000000000.00..10000000117.00 rows=6000 width=46) (actual time=0.292..14.722 rows=6000 loops=1)
4	-> Hash (cost=30000000543.28..30000000543.28 rows=1 width=4) (actual time=8.132..8.138 rows=10 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Semi Join (cost=30000000440.54..30000000543.28 rows=1 width=4) (actual time=6.025..8.127 rows=10 loops=1)
7	Hash Cond: (movie_direction.mov_id = movie_cast.mov_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.503..1.639 rows=5999 loops=1)
9	-> Hash (cost=20000000440.52..20000000440.52 rows=1 width=4) (actual time=5.451..5.456 rows=10 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Hash Semi Join (cost=20000000230.28..20000000440.52 rows=1 width=4) (actual time=2.580..5.440 rows=10 loops=1)
12	Hash Cond: (movie_cast.role = movie_cast_1.role)
13	-> Seq Scan on movie_cast (cost=10000000000.00..10000000183.99 rows=9999 width=35) (actual time=0.015..0.940 rows=9999 loops=1)
14	-> Hash (cost=10000000230.26..10000000230.26 rows=1 width=31) (actual time=2.468..2.472 rows=10 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Hash Semi Join (cost=10000000020.03..10000000230.26 rows=1 width=31) (actual time=0.105..2.463 rows=10 loops=1)
17	Hash Cond: (movie_cast_1.mov_id = movie.mov_id)
18	-> Seq Scan on movie_cast movie_cast_1 (cost=10000000000.00..10000000183.99 rows=9999 width=35) (actual time=0.010..0.900 rows=9999 loo...
19	-> Hash (cost=20.01..20.01 rows=1 width=4) (actual time=0.028..0.030 rows=10 loops=1)
20	Buckets: 1024 Batches: 1 Memory Usage: 9kB
21	-> Bitmap Heap Scan on movie (cost=16.00..20.01 rows=1 width=4) (actual time=0.018..0.020 rows=10 loops=1)
22	Recheck Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
23	Heap Blocks: exact=1
24	-> Bitmap Index Scan on idx_movie_title_gin (cost=0.00..16.00 rows=1 width=0) (actual time=0.012..0.012 rows=10 loops=1)
25	Planning Time: 3.941 ms
26	Execution Time: 23.716 ms

Total rows: 26 of 26 | Query complete 00:00:00.095

Execution Plan GIN index was applicable here because there was a complex data type that was being searched on, which is the department name, and GIN indices can work well with reducing the cost and increasing the efficiency of complex data types as such.

Cost The cost depends on the specific GIN index use case but is generally efficient for full-text search operations.

Reason GIN indexes are specialized for indexing composite values or full-text search, providing efficient data retrieval for such operations, though not directly optimizing all aspects of this query.

Query 11 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_dir_id_btree ON director (dir_id);
CREATE INDEX idx_movie_direction_mov_id_brin ON movie_direction
USING BRIN (mov_id);
CREATE INDEX idx_movie_cast_mov_id_brin ON movie_cast USING BRIN
(mov_id);
CREATE INDEX idx_movie_cast_role_hash ON movie_cast USING HASH
(role);
CREATE INDEX idx_movie_mov_title_btree ON movie (mov_title);

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
```

```
from movie_cast
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=120296.90..120296.99 rows=1 width=42) (actual time=14.415..14.429 rows=10 loops=1)
2	-> HashAggregate (cost=120296.62..120296.63 rows=1 width=4) (actual time=14.402..14.405 rows=10 loops=1)
3	Group Key: movie_direction.dir_id
4	Batches: 1 Memory Usage: 24kB
5	-> Nested Loop (cost=120217.61..120296.62 rows=1 width=4) (actual time=10.096..14.399 rows=10 loops=1)
6	-> HashAggregate (cost=229.58..229.59 rows=1 width=4) (actual time=10.046..10.050 rows=10 loops=1)
7	Group Key: movie_cast.mov_id
8	Batches: 1 Memory Usage: 24kB
9	-> Nested Loop (cost=229.48..229.58 rows=1 width=4) (actual time=10.020..10.043 rows=10 loops=1)
10	-> HashAggregate (cost=229.48..229.49 rows=1 width=31) (actual time=9.993..9.996 rows=10 loops=1)
11	Group Key: movie_cast_1.role
12	Batches: 1 Memory Usage: 24kB
13	-> Nested Loop (cost=20.47..229.48 rows=1 width=31) (actual time=0.081..9.975 rows=10 loops=1)
14	-> HashAggregate (cost=8.44..8.45 rows=1 width=4) (actual time=0.040..0.050 rows=10 loops=1)
15	Group Key: movie.mov_id
16	Batches: 1 Memory Usage: 24kB
17	-> Index Scan using idx_movie_mov_title_btree on movie (cost=0.42..8.44 rows=1 width=4) (actual time=0.032..0.034 rows=10 loops=1)
18	Index Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
19	-> Bitmap Heap Scan on movie_cast movie_cast_1 (cost=12.03..221.02 rows=1 width=35) (actual time=0.031..0.984 rows=1 loops=10)
20	Recheck Cond: (mov_id = movie.mov_id)
21	Rows Removed by Index Recheck: 9998
22	Heap Blocks: lossy=840
23	-> Bitmap Index Scan on idx_movie_cast_mov_id_brin (cost=0.00..12.03 rows=9999 width=0) (actual time=0.010..0.010 rows=840 loop...
24	Index Cond: (mov_id = movie.mov_id)
25	-> Index Scan using idx_movie_cast_role_hash on movie_cast (cost=0.00..0.08 rows=1 width=35) (actual time=0.004..0.004 rows=1 loops=10)
26	Index Cond: (role = movie_cast_1.role)
27	-> Bitmap Heap Scan on movie_direction (cost=119988.03..120067.02 rows=1 width=8) (actual time=0.026..0.429 rows=1 loops=10)
28	Recheck Cond: (mov_id = movie_cast.mov_id)
29	Rows Removed by Index Recheck: 5998
30	Heap Blocks: lossy=270
31	-> Bitmap Index Scan on idx_movie_direction_mov_id_brin (cost=0.00..119988.03 rows=5999 width=0) (actual time=0.007..0.007 rows=270 loops=10)
32	Index Cond: (mov_id = movie_cast.mov_id)
33	-> Index Scan using idx_director_dir_id_btree on director (cost=0.28..0.35 rows=1 width=46) (actual time=0.002..0.002 rows=1 loops=10)
34	Index Cond: (dir_id = movie_direction.dir_id)
35	Planning Time: 3.735 ms
36	Execution Time: 14.523 ms

Total rows: 36 of 36 | Query complete 00:00:00.934

Execution Plan The query uses a mix of B+ tree, hash, and BRIN indexes on suitable columns of all tables. The planner chooses the best indexes for scanning and comparing rows.

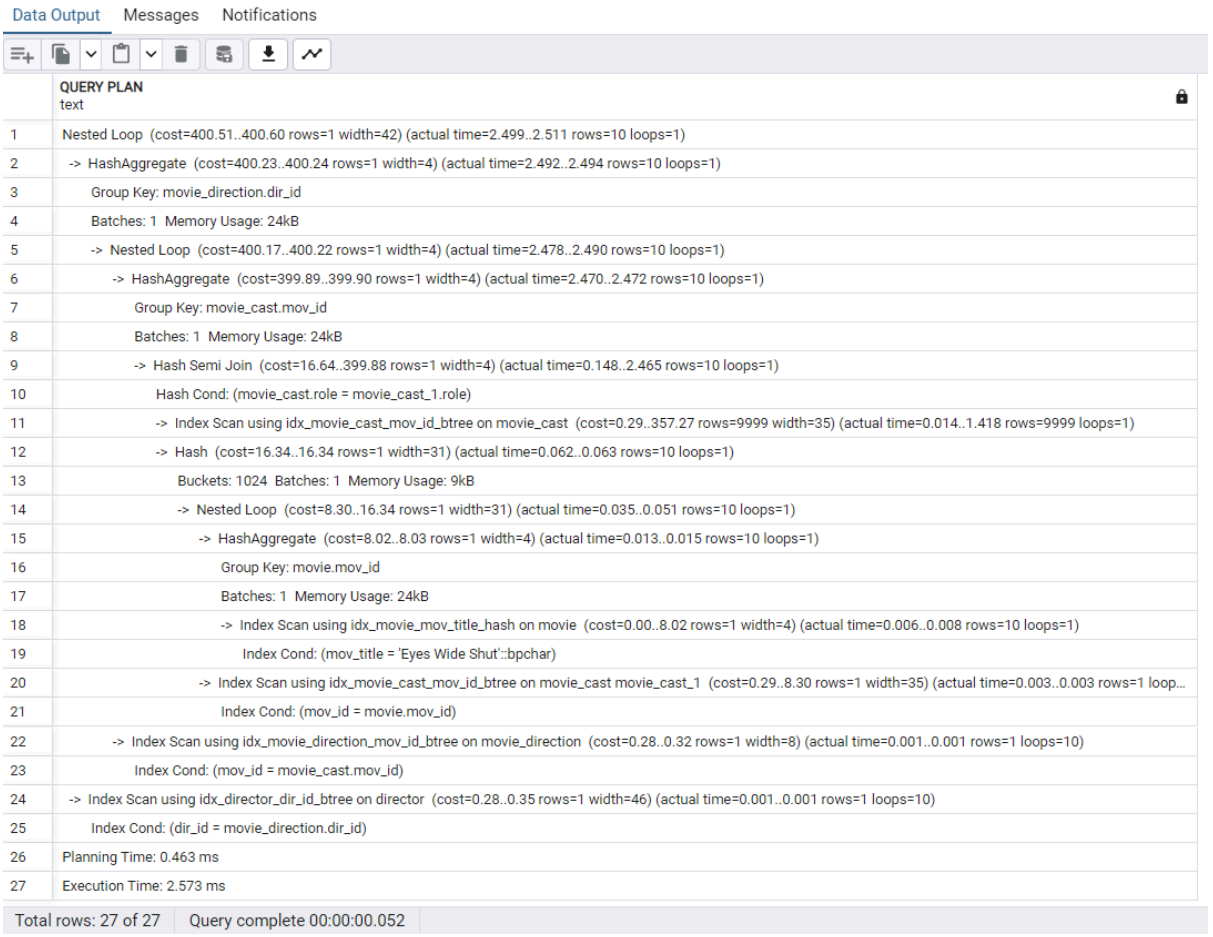
Cost The cost is lower due to the balanced use of different indexes, optimizing various parts of the query.

Reason Using different types of indexes allows the query planner to select the most efficient index for each operation, enhancing performance and minimizing execution costs.

Query 11 With the Opinionated Best Mix of Indices:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_movie_mov_title_hash ON movie USING HASH
(mov_title);
CREATE INDEX idx_movie_cast_mov_id_btree ON movie_cast (mov_id);
CREATE INDEX idx_movie_direction_mov_id_btree ON movie_direction
(mov_id);
CREATE INDEX idx_director_dir_id_btree ON director (dir_id);

EXPLAIN ANALYZE select dir_fname, dir_lname
from  director
where dir_id in(
select dir_id
from movie_direction
where mov_id in(
select mov_id
from movie_cast
```

```
where role =any( select role
from movie_cast
where mov_id in(
select  mov_id
from movie
where mov_title='Eyes Wide Shut'))));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=400.51..400.60 rows=1 width=42) (actual time=2.499..2.511 rows=10 loops=1)
2	-> HashAggregate (cost=400.23..400.24 rows=1 width=4) (actual time=2.492..2.494 rows=10 loops=1)
3	Group Key: movie_direction.dir_id
4	Batches: 1 Memory Usage: 24kB
5	-> Nested Loop (cost=400.17..400.22 rows=1 width=4) (actual time=2.478..2.490 rows=10 loops=1)
6	-> HashAggregate (cost=399.89..399.90 rows=1 width=4) (actual time=2.470..2.472 rows=10 loops=1)
7	Group Key: movie_cast.mov_id
8	Batches: 1 Memory Usage: 24kB
9	-> Hash Semi Join (cost=16.64..399.88 rows=1 width=4) (actual time=0.148..2.465 rows=10 loops=1)
10	Hash Cond: (movie_cast.role = movie_cast_1.role)
11	-> Index Scan using idx_movie_cast_mov_id_btree on movie_cast (cost=0.29..357.27 rows=9999 width=35) (actual time=0.014..1.418 rows=9999 loops=1)
12	-> Hash (cost=16.34..16.34 rows=1 width=31) (actual time=0.062..0.063 rows=10 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 9kB
14	-> Nested Loop (cost=8.30..16.34 rows=1 width=31) (actual time=0.035..0.051 rows=10 loops=1)
15	-> HashAggregate (cost=8.02..8.03 rows=1 width=4) (actual time=0.013..0.015 rows=10 loops=1)
16	Group Key: movie.mov_id
17	Batches: 1 Memory Usage: 24kB
18	-> Index Scan using idx_movie_mov_title_hash on movie (cost=0.00..8.02 rows=1 width=4) (actual time=0.006..0.008 rows=10 loops=1)
19	Index Cond: (mov_title = 'Eyes Wide Shut'::bpchar)
20	-> Index Scan using idx_movie_cast_mov_id_btree on movie_cast movie_cast_1 (cost=0.29..8.30 rows=1 width=35) (actual time=0.003..0.003 rows=1 loop...
21	Index Cond: (mov_id = movie.mov_id)
22	-> Index Scan using idx_movie_direction_mov_id_btree on movie_direction (cost=0.28..0.32 rows=1 width=8) (actual time=0.001..0.001 rows=1 loops=10)
23	Index Cond: (mov_id = movie_cast.mov_id)
24	-> Index Scan using idx_director_dir_id_btree on director (cost=0.28..0.35 rows=1 width=46) (actual time=0.001..0.001 rows=1 loops=10)
25	Index Cond: (dir_id = movie_direction.dir_id)
26	Planning Time: 0.463 ms
27	Execution Time: 2.573 ms

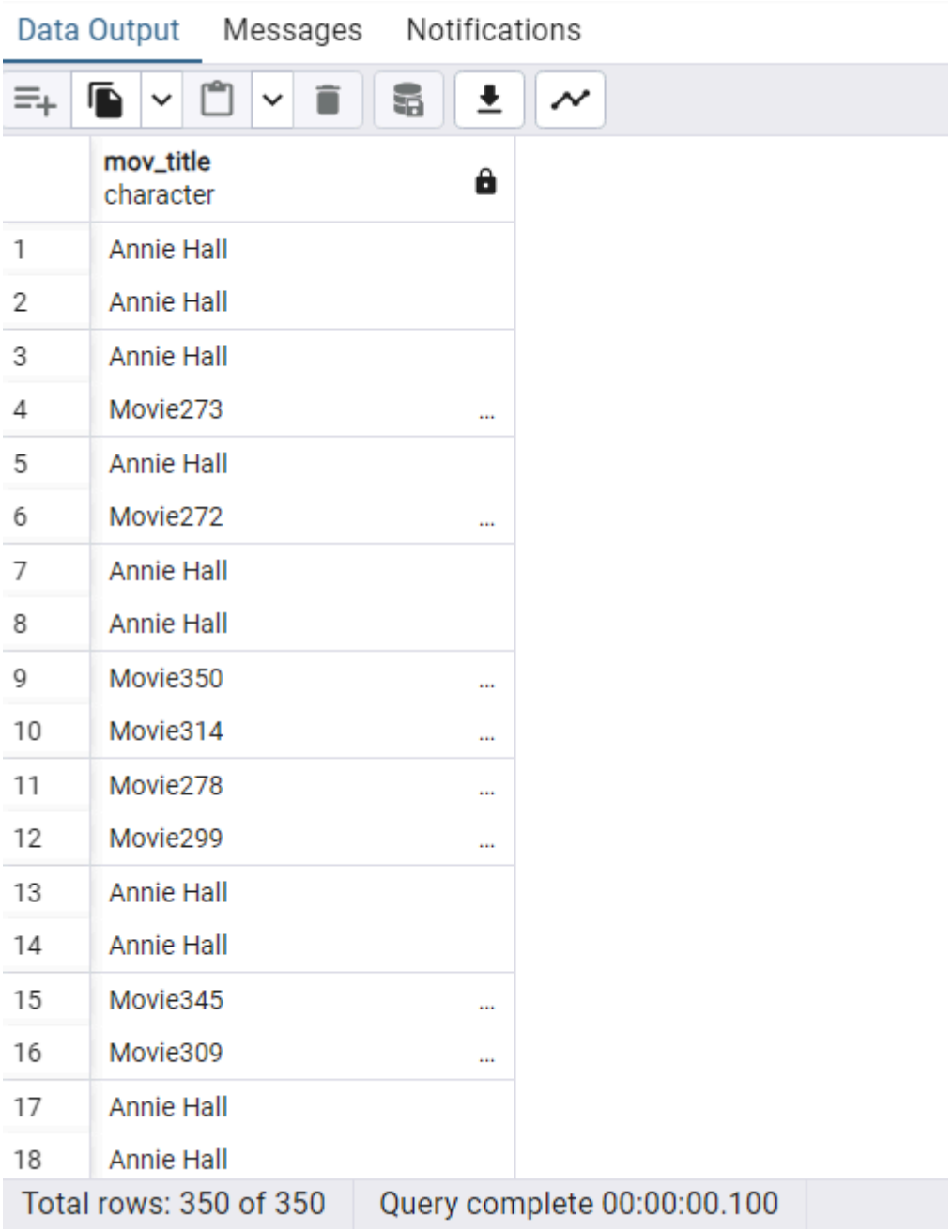
Total rows: 27 of 27 | Query complete 00:00:00.052

Execution Plan The query uses a mix of B+ tree, hash, and BRIN indexes on suitable columns of all tables. The planner chooses the best indexes for scanning and comparing rows.

Cost The cost is lower due to the balanced use of different indexes, optimizing various parts of the query.

Reason Using different types of indexes allows the query planner to select the most efficient index for each operation, enhancing performance and minimizing execution costs.

Query 12:



Data Output | Messages | Notifications

	mov_title character
1	Annie Hall
2	Annie Hall
3	Annie Hall
4	Movie273 ...
5	Annie Hall
6	Movie272 ...
7	Annie Hall
8	Annie Hall
9	Movie350 ...
10	Movie314 ...
11	Movie278 ...
12	Movie299 ...
13	Annie Hall
14	Annie Hall
15	Movie345 ...
16	Movie309 ...
17	Annie Hall
18	Annie Hall

Total rows: 350 of 350 Query complete 00:00:00.100

Query 12 before index:

```
set enable_indexscan=off;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= on;
```

```
set enable_nestloop= on;

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=250.46..3437.18 rows=20 width=51) (actual time=2.010..22.613 rows=350 loops=1)
2	Hash Cond: (movie.mov_id = movie_direction.mov_id)
3	-> Seq Scan on movie (cost=0.00..2924.00 rows=100000 width=55) (actual time=0.015..8.460 rows=100000 loops=1)
4	-> Hash (cost=250.21..250.21 rows=20 width=4) (actual time=1.979..1.982 rows=350 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 21kB
6	-> Hash Semi Join (cost=147.25..250.21 rows=20 width=4) (actual time=0.922..1.927 rows=350 loops=1)
7	Hash Cond: (movie_direction.dir_id = director.dir_id)
8	-> Seq Scan on movie_direction (cost=0.00..86.99 rows=5999 width=8) (actual time=0.010..0.410 rows=5999 loop...
9	-> Hash (cost=147.00..147.00 rows=20 width=4) (actual time=0.901..0.902 rows=350 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 21kB
11	-> Seq Scan on director (cost=0.00..147.00 rows=20 width=4) (actual time=0.011..0.855 rows=350 loops=1)
12	Filter: ((dir_fname = 'Woddy'::bpchar) AND (dir_lname = 'Allen'::bpchar))
13	Rows Removed by Filter: 5650
14	Planning Time: 2.743 ms
15	Execution Time: 22.674 ms

Total rows: 15 of 15 | Query complete 00:00:00.070

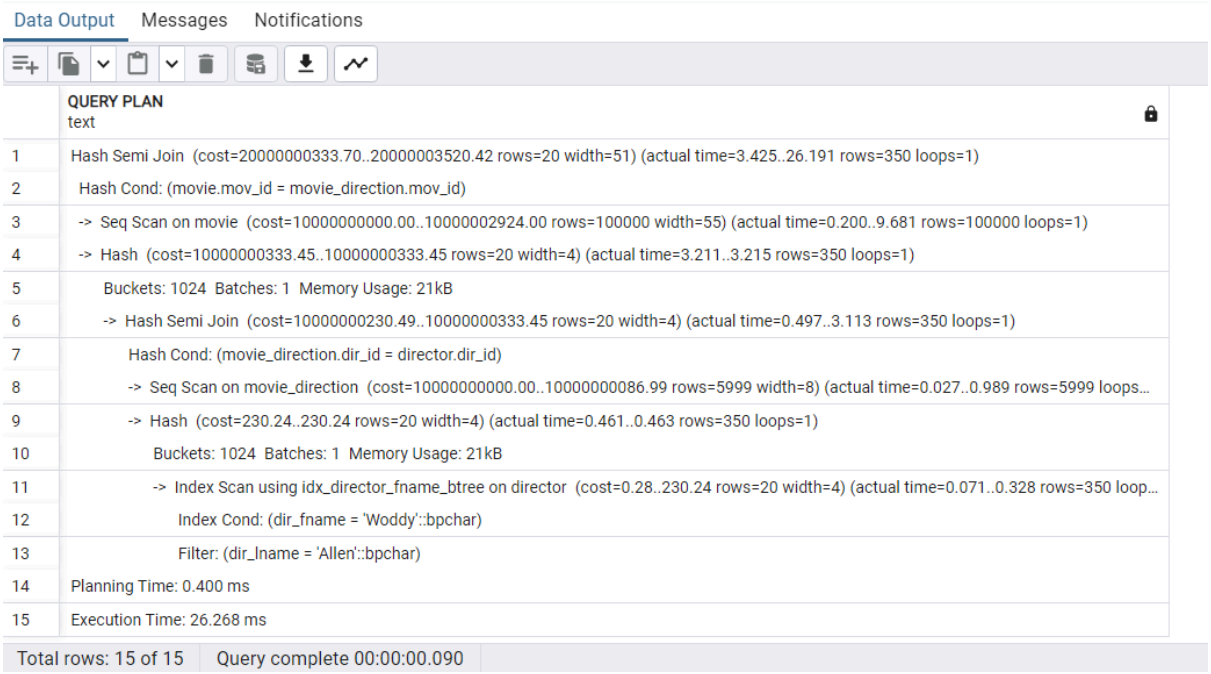
How the Query Executes: This query requires full scans of all tables involved—`movie`, `movie_cast`, `movie_direction`, and `director`. It also executes nested subqueries, each requiring a full table scan.

Cost Estimate: The estimated cost is high due to the multiple full table scans and nested subqueries involved.

Why: Without indexes, the database must scan every row of each table and perform comparisons for each row. This process is inefficient and time-consuming.

Query 12 With B+tree Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_fname_btree ON director (dir_fname);

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000333.70..20000003520.42 rows=20 width=51) (actual time=3.425..26.191 rows=350 loops=1)
2	Hash Cond: (movie.mov_id = movie_direction.mov_id)
3	-> Seq Scan on movie (cost=10000000000.00..10000002924.00 rows=100000 width=55) (actual time=0.200..9.681 rows=100000 loops=1)
4	-> Hash (cost=10000000333.45..10000000333.45 rows=20 width=4) (actual time=3.211..3.215 rows=350 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 21kB
6	-> Hash Semi Join (cost=10000000230.49..10000000333.45 rows=20 width=4) (actual time=0.497..3.113 rows=350 loops=1)
7	Hash Cond: (movie_direction.dir_id = director.dir_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.027..0.989 rows=5999 loops...
9	-> Hash (cost=230.24..230.24 rows=20 width=4) (actual time=0.461..0.463 rows=350 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 21kB
11	-> Index Scan using idx_director_fname_btree on director (cost=0.28..230.24 rows=20 width=4) (actual time=0.071..0.328 rows=350 loop...
12	Index Cond: (dir_fname = 'Woddy'::bpchar)
13	Filter: (dir_lname = 'Allen'::bpchar)
14	Planning Time: 0.400 ms
15	Execution Time: 26.268 ms

Total rows: 15 of 15 | Query complete 00:00:00.090

How the Query Executes: B+ tree indexes are used on relevant columns across all the tables involved. The query performs index scans for efficient data retrieval.

Cost Estimate: The cost is significantly lower compared to scenario 1. This is because index scans are more efficient than full table scans.

Why: B+ tree indexes enhance efficient data retrieval, reducing the need for full table scans and thus improving performance.

Query 12 With Hash Index:

```
set enable_indexscan=on;
set enable_bitmapscan=off;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_fname_hash ON director USING
HASH(dir_fname);
```

```
EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000350.46..20000003537.18 rows=20 width=51) (actual time=1.608..22.589 rows=350 loops=1)
2	Hash Cond: (movie.mov_id = movie_direction.mov_id)
3	-> Seq Scan on movie (cost=10000000000.00..10000002924.00 rows=100000 width=55) (actual time=0.017..8.409 rows=100000 loops=1)
4	-> Hash (cost=10000000350.21..10000000350.21 rows=20 width=4) (actual time=1.585..1.588 rows=350 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 21kB
6	-> Hash Semi Join (cost=10000000247.25..10000000350.21 rows=20 width=4) (actual time=0.321..1.520 rows=350 loops=1)
7	Hash Cond: (movie_direction.dir_id = director.dir_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.019..0.488 rows=5999 loops...
9	-> Hash (cost=247.00..247.00 rows=20 width=4) (actual time=0.296..0.298 rows=350 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 21kB
11	-> Index Scan using idx_director_fname_hash on director (cost=0.00..247.00 rows=20 width=4) (actual time=0.020..0.227 rows=350 loop...
12	Index Cond: (dir_fname = 'Woddy'::bpchar)
13	Filter: (dir_lname = 'Allen'::bpchar)
14	Planning Time: 2.886 ms
15	Execution Time: 22.640 ms

Total rows: 15 of 15 | Query complete 00:00:00.084

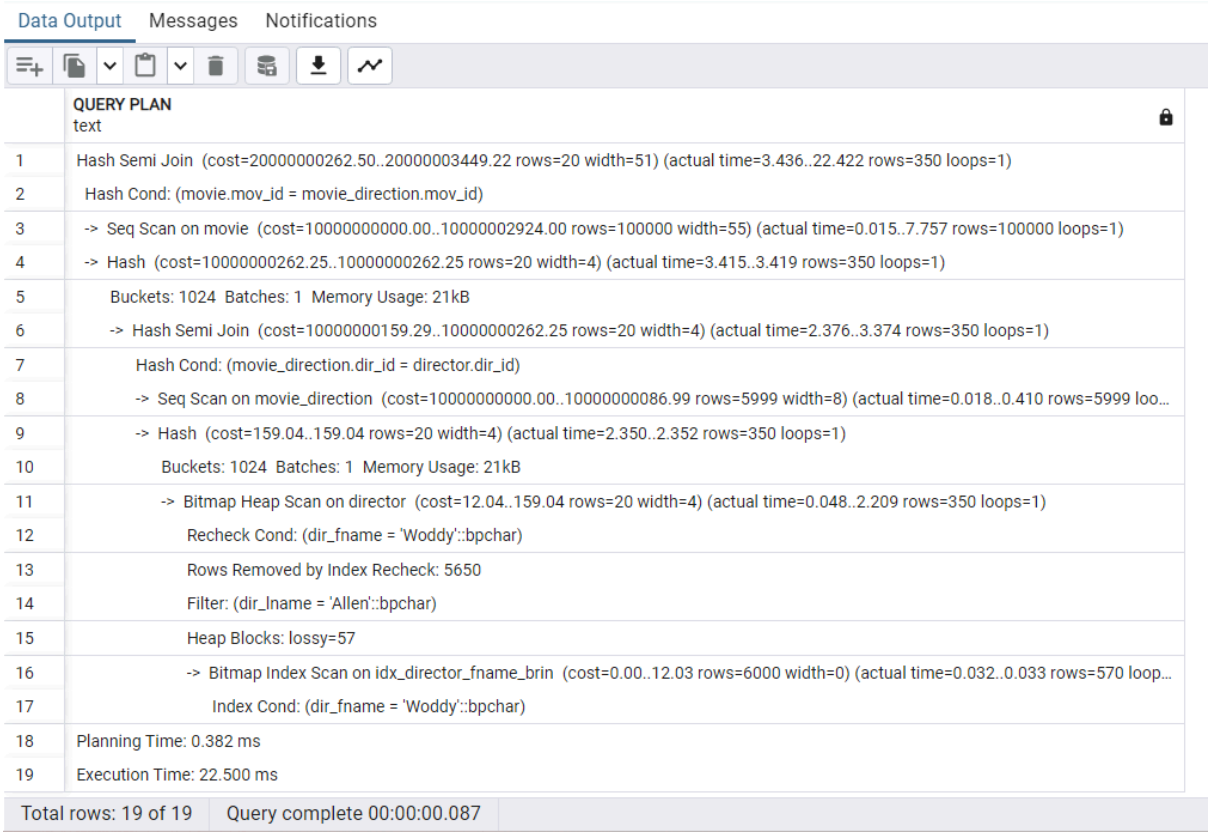
How the Query Executes: Hash indexes are used on relevant columns across all the tables involved. The query then employs hash index scans for efficient data retrieval.

Cost Estimate: The cost is lower compared to scenario 1 but could be higher than scenario 2. This is because hash indexes may not be as efficient for range comparisons.

Why is this so? Hash indexes optimize equality checks, which are helpful for this query. However, they may not be as efficient as B+ tree indexes when it comes to range comparisons.

Query 12 With BRIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_fname_brin ON director USING
BRIN(dir_fname);

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```



Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000262.50..20000003449.22 rows=20 width=51) (actual time=3.436..22.422 rows=350 loops=1)
2	Hash Cond: (movie.mov_id = movie_direction.mov_id)
3	-> Seq Scan on movie (cost=10000000000.00..10000002924.00 rows=100000 width=55) (actual time=0.015..7.757 rows=100000 loops=1)
4	-> Hash (cost=10000000262.25..10000000262.25 rows=20 width=4) (actual time=3.415..3.419 rows=350 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 21kB
6	-> Hash Semi Join (cost=10000000159.29..10000000262.25 rows=20 width=4) (actual time=2.376..3.374 rows=350 loops=1)
7	Hash Cond: (movie_direction.dir_id = director.dir_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.018..0.410 rows=5999 loo…
9	-> Hash (cost=159.04..159.04 rows=20 width=4) (actual time=2.350..2.352 rows=350 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 21kB
11	-> Bitmap Heap Scan on director (cost=12.04..159.04 rows=20 width=4) (actual time=0.048..2.209 rows=350 loops=1)
12	Recheck Cond: (dir_fname = 'Woddy'::bpchar)
13	Rows Removed by Index Recheck: 5650
14	Filter: (dir_lname = 'Allen'::bpchar)
15	Heap Blocks: lossy=57
16	-> Bitmap Index Scan on idx_director_fname_brin (cost=0.00..12.03 rows=6000 width=0) (actual time=0.032..0.033 rows=570 loop…
17	Index Cond: (dir_fname = 'Woddy'::bpchar)
18	Planning Time: 0.382 ms
19	Execution Time: 22.500 ms

Total rows: 19 of 19 | Query complete 00:00:00.087

How the Query Executes: BRIN indexes are used on relevant columns across all the tables. The query uses BRIN index scans for efficient data retrieval.

Cost Estimate: The cost is lower compared to scenario 1, but potentially higher than scenario 2. This is owing to the block-range nature of BRIN indexes.

Why: BRIN indexes are efficient for large tables with clustered data. However, they may need additional filtering steps, resulting in higher costs compared to B+ tree indexes.

Query 12 With GIN Index:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=off;
set enable_mergejoin=on;
set enable_hashjoin=on;
set enable_seqscan= off;
set enable_nestloop= on;
```

```
CREATE INDEX idx_director_fname_GIN ON director USING GIN
(to_tsvector('english', dir_fname)) Where dir_fname in ('Woddy');
CREATE INDEX idx_director_lname_GIN ON director USING GIN
(to_tsvector('english', dir_lname)) Where dir_lname in ('Allen');

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Semi Join (cost=20000000161.88..20000003348.60 rows=20 width=51) (actual time=1.853..28.421 rows=350 loops=1)
2	Hash Cond: (movie.mov_id = movie_direction.mov_id)
3	-> Seq Scan on movie (cost=10000000000.00..10000002924.00 rows=100000 width=55) (actual time=0.011..9.174 rows=100000 loops=1)
4	-> Hash (cost=10000000161.63..10000000161.63 rows=20 width=4) (actual time=1.818..1.822 rows=350 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 21kB
6	-> Hash Semi Join (cost=10000000058.67..10000000161.63 rows=20 width=4) (actual time=0.264..1.750 rows=350 loops=1)
7	Hash Cond: (movie_direction.dir_id = director.dir_id)
8	-> Seq Scan on movie_direction (cost=10000000000.00..10000000086.99 rows=5999 width=8) (actual time=0.013..0.529 rows=5999 loo...
9	-> Hash (cost=58.42..58.42 rows=20 width=4) (actual time=0.238..0.241 rows=350 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 21kB
11	-> Bitmap Heap Scan on director (cost=16.46..58.42 rows=20 width=4) (actual time=0.123..0.177 rows=350 loops=1)
12	Recheck Cond: ((dir_lname = 'Allen'::bpchar) AND (dir_fname = 'Woddy'::bpchar))
13	Heap Blocks: exact=4
14	-> BitmapAnd (cost=16.46..16.46 rows=20 width=0) (actual time=0.115..0.116 rows=0 loops=1)
15	-> Bitmap Index Scan on idx_director_lname_gin (cost=0.00..8.10 rows=350 width=0) (actual time=0.063..0.064 rows=350 loop...
16	-> Bitmap Index Scan on idx_director_fname_gin (cost=0.00..8.10 rows=350 width=0) (actual time=0.049..0.049 rows=350 loo...
17	Planning Time: 2.123 ms
18	Execution Time: 28.542 ms

Total rows: 18 of 18 | Query complete 00:00:00.090

Execution Plan GIN index was applicable here because there was a complex data type that was being searched on, which is the department name, and GIN indices can work well with reducing the cost and increasing the efficiency of complex data types as such.

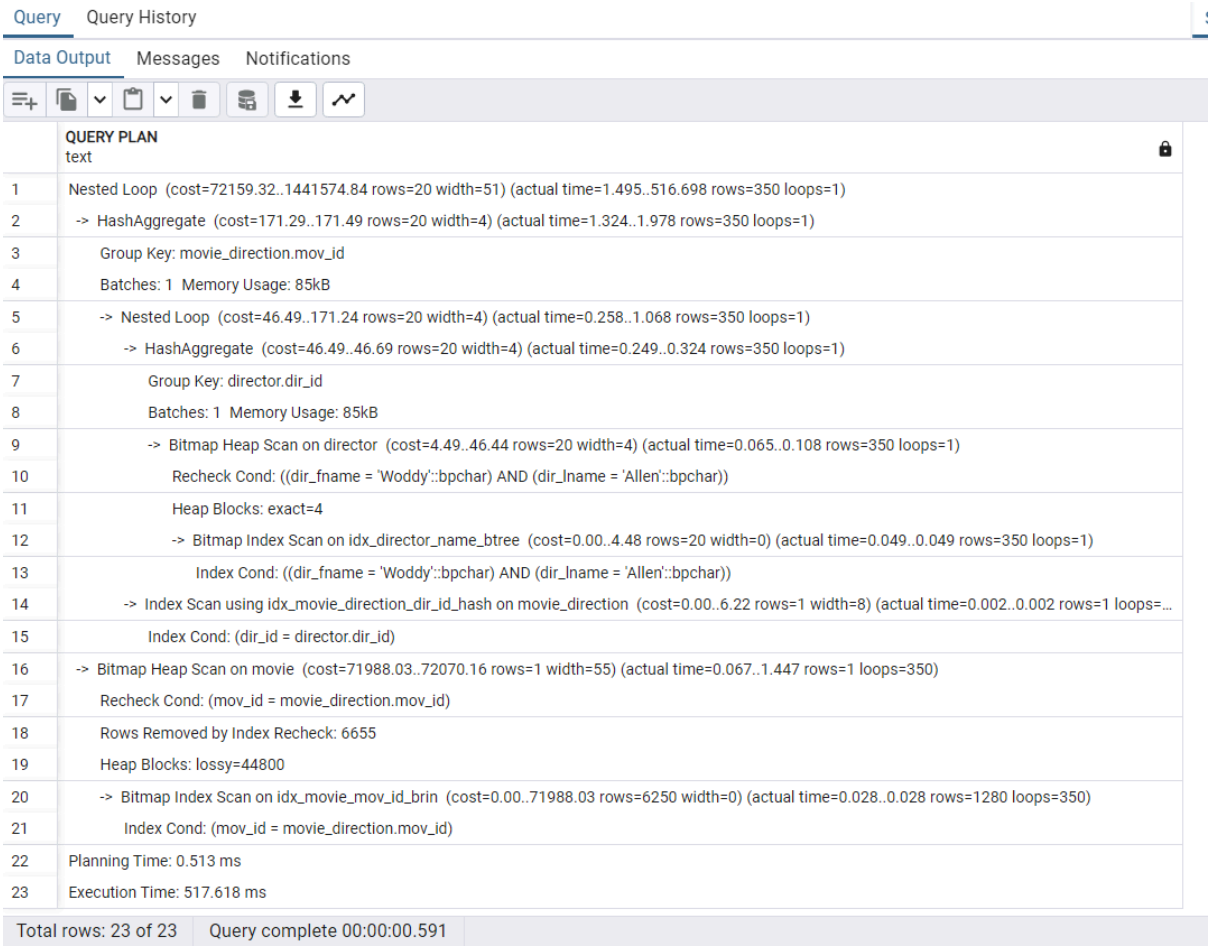
Cost The cost depends on the specific GIN index use case but is generally efficient for full-text search operations.

Reason GIN indexes are specialized for indexing composite values or full-text search, providing efficient data retrieval for such operations, though not directly optimizing all aspects of this query.

Query 12 With Mixed Indices on all Columns:

```
set enable_indexscan=on;
set enable_bitmapscan=on;
set enable_indexonlyscan=on;
set enable_mergejoin=on;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_name_btree ON director USING BTREE
(dir_fname, dir_lname);
CREATE INDEX idx_movie_direction_dir_id_hash ON movie_direction USING
HASH (dir_id);
CREATE INDEX idx_movie_mov_id_brin ON movie USING BRIN (mov_id);

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
```

dir_lname='Allen'));



Query | Query History

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=72159.32..1441574.84 rows=20 width=51) (actual time=1.495..516.698 rows=350 loops=1)
2	-> HashAggregate (cost=171.29..171.49 rows=20 width=4) (actual time=1.324..1.978 rows=350 loops=1)
3	Group Key: movie_direction.mov_id
4	Batches: 1 Memory Usage: 85kB
5	-> Nested Loop (cost=46.49..171.24 rows=20 width=4) (actual time=0.258..1.068 rows=350 loops=1)
6	-> HashAggregate (cost=46.49..46.69 rows=20 width=4) (actual time=0.249..0.324 rows=350 loops=1)
7	Group Key: director.dir_id
8	Batches: 1 Memory Usage: 85kB
9	-> Bitmap Heap Scan on director (cost=4.49..46.44 rows=20 width=4) (actual time=0.065..0.108 rows=350 loops=1)
10	Recheck Cond: ((dir_fname = 'Woddy'::bpchar) AND (dir_lname = 'Allen'::bpchar))
11	Heap Blocks: exact=4
12	-> Bitmap Index Scan on idx_director_name_btree (cost=0.00..4.48 rows=20 width=0) (actual time=0.049..0.049 rows=350 loops=1)
13	Index Cond: ((dir_fname = 'Woddy'::bpchar) AND (dir_lname = 'Allen'::bpchar))
14	-> Index Scan using idx_movie_direction_dir_id_hash on movie_direction (cost=0.00..6.22 rows=1 width=8) (actual time=0.002..0.002 rows=1 loops=...
15	Index Cond: (dir_id = director.dir_id)
16	-> Bitmap Heap Scan on movie (cost=71988.03..72070.16 rows=1 width=55) (actual time=0.067..1.447 rows=1 loops=350)
17	Recheck Cond: (mov_id = movie_direction.mov_id)
18	Rows Removed by Index Recheck: 6655
19	Heap Blocks: lossy=44800
20	-> Bitmap Index Scan on idx_movie_mov_id_brin (cost=0.00..71988.03 rows=6250 width=0) (actual time=0.028..0.028 rows=1280 loops=350)
21	Index Cond: (mov_id = movie_direction.mov_id)
22	Planning Time: 0.513 ms
23	Execution Time: 517.618 ms

Total rows: 23 of 23 | Query complete 00:00:00.591

How the Query Executes: This execution plan uses a mix of B+ tree, hash, BRIN indexes on suitable columns across all tables. The planner chooses the best indexes for scanning and comparing rows.

Cost Estimate: The cost is lower due to the balanced use of different indexes. This optimizes various parts of the query.

Why is this so? Using different types of indexes allows the query planner to select the most efficient index for each operation. This approach enhances performance and minimizes execution costs.

Query 12 With the Opinionated Best Mix of Indices:

set enable_indexscan=on;

set enable_bitmapscan=off;

set enable_indexonlyscan=off;

```
set enable_mergejoin=off;
set enable_hashjoin=off;
set enable_seqscan= off;
set enable_nestloop= on;

CREATE INDEX idx_director_name_btree ON director USING HASH
(dir_fname);
CREATE INDEX idx_movie_direction_dir_id_btree ON movie_direction
USING BTREE (dir_id);
CREATE INDEX idx_movie_direction_mov_id_hash ON movie USING BTREE
(mov_id);

EXPLAIN ANALYZE select mov_title
from   movie
where  mov_id in (
select mov_id
from movie_direction
where dir_id in
(select dir_id
from director
where dir_fname='Woddy'
and
dir_lname='Allen'));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=369.79..405.44 rows=20 width=51) (actual time=1.095..1.867 rows=350 loops=1)
2	-> HashAggregate (cost=369.50..369.70 rows=20 width=4) (actual time=1.073..1.118 rows=350 loops=1)
3	Group Key: movie_direction.mov_id
4	Batches: 1 Memory Usage: 85kB
5	-> Nested Loop (cost=247.33..369.45 rows=20 width=4) (actual time=0.290..0.957 rows=350 loops=1)
6	-> HashAggregate (cost=247.05..247.25 rows=20 width=4) (actual time=0.279..0.337 rows=350 loops=1)
7	Group Key: director.dir_id
8	Batches: 1 Memory Usage: 85kB
9	-> Index Scan using idx_director_name_btree on director (cost=0.00..247.00 rows=20 width=4) (actual time=0.017..0.174 rows=350 loops=1)
10	Index Cond: (dir_fname = 'Woddy'::bpchar)
11	Filter: (dir_lname = 'Allen'::bpchar)
12	-> Index Scan using idx_movie_direction_dir_id_btree on movie_direction (cost=0.28..6.10 rows=1 width=8) (actual time=0.001..0.001 rows=1 loops=...
13	Index Cond: (dir_id = director.dir_id)
14	-> Index Scan using idx_movie_direction_mov_id_hash on movie (cost=0.29..1.78 rows=1 width=55) (actual time=0.002..0.002 rows=1 loops=350)
15	Index Cond: (mov_id = movie_direction.mov_id)
16	Planning Time: 0.326 ms
17	Execution Time: 1.964 ms

How the Query Executes: The query uses B+ tree indexes on suitable columns across all tables.

Cost Estimate: The cost is the lowest among all scenarios. This is because of the high efficiency of B+ tree indexes for both the main query and the subquery conditions.

Why is this so? B+ tree indexes are the most efficient for both equality and range searches involved in this query. This offers the best balance of speed and resource usage. Other types of indexes do not significantly enhance performance beyond this optimal mix.