German University in Cairo

Media Engineering and Technology

A report about

Data Base MiniProject2

Done by:

Arwa Tawfik Aboukhadra T-14 43-12603
Donia Ali T-14 43-8354
Reem Ayman T-14 43-11979
Dalia Walid T-11 43-3745
Sarah Farrag T-14 43-15601

Supervised by:

prof. WaelAbuElsadaat

I. Table Of Contents

‘

Schema1

Data insertion modifications:

```
public static void populateDepartment(Connection conn) {

if (insertDepartment(1, "CS" + 1, 1, conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(2, "CS" + 2, 2, conn) == 0) {
System.err.println("insertion of record " + 2 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(3, "CS" + 3, 3, conn) == 0) {
System.err.println("insertion of record " + 3 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(4, "CS" + 4, 4, conn) == 0) {
System.err.println("insertion of record " + 4 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(5, "CS" + 5, 5, conn) == 0) {
System.err.println("insertion of record " + 5 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(6, "CS" + 6, 6, conn) == 0) {
System.err.println("insertion of record " + 6 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(7, "CS" + 7, 7, conn) == 0) {
System.err.println("insertion of record " + 7 + " failed");

} else
```

```
System.out.println("insertion was successful");
if (insertDepartment(8, "CS" + 8, 8, conn) == 0) {
System.err.println("insertion of record " + 8 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(9, "CS" + 9, 9, conn) == 0) {
System.err.println("insertion of record " + 9 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(10, "CS" + 10, 10, conn) == 0) {
System.err.println("insertion of record " + 10 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(11, "Math" + 1, 11, conn) == 0) {
System.err.println("insertion of record " + 11 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(12, "Math" + 2, 12, conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(13, "Math" + 3, 13, conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(14, "Math" + 4, 14, conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");

} else
System.out.println("insertion was successful");
if (insertDepartment(15, "Math" + 5, 15, conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");
```

```
} else
System.out.println("insertion was successful");

}

public static void populateInstructor(Connection conn) {
for (int i = 1; i < 12; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");

}
for (int i = 12; i < 24; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 2, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");

}
for (int i = 24; i < 36; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 3, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 36; i < 48; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 4, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");

}
for (int i = 48; i < 60; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 5, conn) == 0) {
```

```
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 60; i < 72; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 6, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 72; i < 84; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 7, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}

for (int i = 84; i < 96; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 8, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 96; i < 108; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 9, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 108; i < 120; i++) {
if (insertInstructor(i, "Name" + i, i, "CS" + 10, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
```

```
System.out.println("insertion was successful");
}
for (int i = 120; i < 132; i++) {
if (insertInstructor(i, "Name" + i, i, "Math" + 1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 132; i < 144; i++) {
if (insertInstructor(i, "Name" + i, i, "Math" + 2, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 144; i < 156; i++) {
if (insertInstructor(i, "Name" + i, i, "Math" + 3, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 156; i < 168; i++) {
if (insertInstructor(i, "Name" + i, i, "Math" + 4, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 168; i < 180; i++) {
if (insertInstructor(i, "Name" + i, i, "Math" + 5, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}

}
```

```
public static void populateClassroom(Connection conn) {
for (int i = 1; i < 10000; i++) {
if (insertClassroom(i, i, 100 +(int)(Math.random()*30)+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

@SuppressWarnings("deprecation")
public static void populateTimeSlot(Connection conn) {
for (int i = 1; i < 10000; i++) {
if (insertTimeSlot(i, "day" + i, new Time(12, 0, 0), new Time(13, 0, 0), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateStudent(Connection conn) {
for (int i = 1; i < 500; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 1, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 500; i < 1000; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 2,i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 1000; i < 1500; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 3,i%179+1, conn) == 0) {
```

```
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 1500; i < 2000; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 4, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 2000; i < 2500; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 5, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 2500; i < 3000; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 6, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 3000; i < 3500; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 7, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 3500; i < 4000; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 8, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
```

```
}
for (int i = 4000; i < 4500; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 9,i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 4500; i < 5000; i++) {
if (insertStudent(i, "name" + i, i, "CS" + 10,i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 5000; i < 5500; i++) {
if (insertStudent(i, "name" + i, i, "Math" +1 , i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 5500; i < 6000; i++) {
if (insertStudent(i, "name" + i, i, "Math" + 2, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 6000; i < 6500; i++) {
if (insertStudent(i, "name" + i, i, "Math" + 3, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 6500; i < 7000; i++) {
if (insertStudent(i, "name" + i, i, "Math" + 4, i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
```

```
break;
} else
System.out.println("insertion was successful");
}
for (int i = 7000; i < 7500; i++) {
if (insertStudent(i, "name" + i, i, "Math" + 5,i%179+1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateCourse(Connection conn) {
for (int i = 1; i < 15; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 15; i < 30; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 2, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 30; i < 45; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 3, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 45; i < 60; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 4, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
```

```
} else
System.out.println("insertion was successful");
}
for (int i = 60; i < 75; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 5, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 75; i < 90; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 6, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 90; i < 105; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 7, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 105; i < 120; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 8, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 120; i < 135; i++) {
if (insertCourse(i, "CSEN" + i, i, "CS" + 9, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 135; i < 150; i++) {
```

```
if (insertCourse(i, "CSEN" + i, i, "CS" + 10, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 150; i < 165; i++) {
if (insertCourse(i, "CSEN" + i, i, "Math" + 1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 165; i < 180; i++) {
if (insertCourse(i, "CSEN" + i, i, "Math" + 2, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 180; i < 195; i++) {
if (insertCourse(i, "CSEN" + i, i, "Math" + 3, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 195; i < 210; i++) {
if (insertCourse(i, "CSEN" + i, i, "Math" + 4, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
for (int i = 210; i < 225; i++) {
if (insertCourse(i, "CSEN" + i, i, "Math" + 5, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
```

```
System.out.println("insertion was successful");
}

}

public static void populatePrerequiste(Connection conn) {
for (int i = 1; i < 225; i++) {
if (insertPrerequiste(i, i, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateSection(Connection conn) {
int j = 1;
for (int i = 1; i < 180; i++) {
if (insertSection(i, i, 2019, i, i, j, j, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
j++;
}
for (int i = 180; i < 225; i++) {
if (insertSection(i, i, 2018, i-150, i, j, j, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
j++;
}
}

public static void populateTakes(Connection conn) {
double j = 0.7;
for (int i = 1; i <7500; i++) {
if (j >= 5)
```

```
j = 0.7;
if (insertTakes(i, i%224+1, j, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
j += 0.3;
}
}

public static void populateSectionTime(Connection conn) {
for (int i = 1; i < 225; i++) {
if (insertSectionTime(i, i, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}
```

Query 1

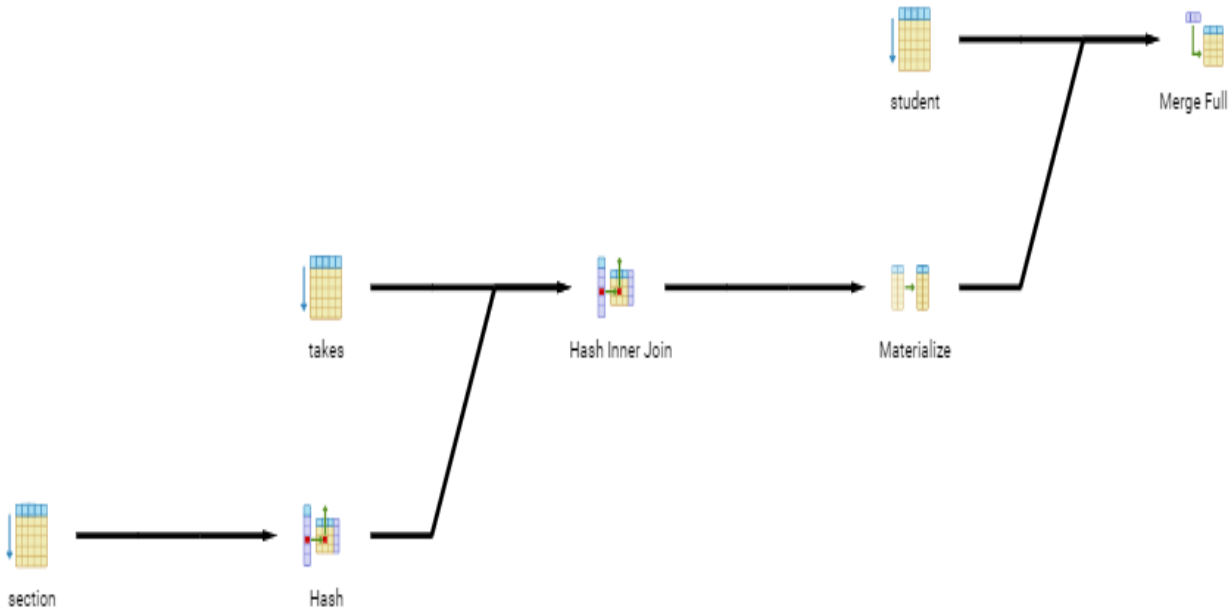
select * from (select * from student where department = 'CS1') as CS1_student natural full outer join (select * from takes t inner join section s on t.section_id = s.section_id where semester = 1 and year = 2019) as sem1_student;

Without Index:

Explain analyze output:

Query Editor | Query History | Notifications | Messages | Data Output | Explain

	QUERY PLAN text
1	Merge Full Join (cost=5.38..708.08 rows=33433 width=64) (actual time=0.395..16.136 rows=16467 loops=1)
2	-> Seq Scan on student (cost=0.00..148.74 rows=499 width=24) (actual time=0.031..2.150 rows=499 loops=1)
3	Filter: ((department)::text = 'CS1'::text)
4	Rows Removed by Filter: 7000
5	-> Materialize (cost=5.38..141.59 rows=67 width=40) (actual time=0.360..5.405 rows=15969 loops=1)
6	-> Hash Join (cost=5.38..141.43 rows=67 width=40) (actual time=0.353..3.932 rows=33 loops=1)
7	Hash Cond: (t.section_id = s.section_id)
8	-> Seq Scan on takes t (cost=0.00..115.99 rows=7499 width=12) (actual time=0.101..2.355 rows=7499 loops=1)
9	-> Hash (cost=5.36..5.36 rows=2 width=28) (actual time=0.069..0.075 rows=1 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on section s (cost=0.00..5.36 rows=2 width=28) (actual time=0.025..0.059 rows=1 loops=1)
12	Filter: ((semester = 1) AND (year = 2019))
13	Rows Removed by Filter: 223
14	Planning Time: 0.599 ms
15	Execution Time: 17.675 ms



"Merge Full Join (cost=5.38..708.08 rows=33433 width=64) (actual time=0.395..16.136 rows=16467 loops=1)"

"Execution Time: 17.675 ms"

"Execution Time: 18.184 ms"

"Execution Time: 12.453 ms"

Explanation:

As we notice from query plan and explain analyze output that seqscan is used and no index is applied

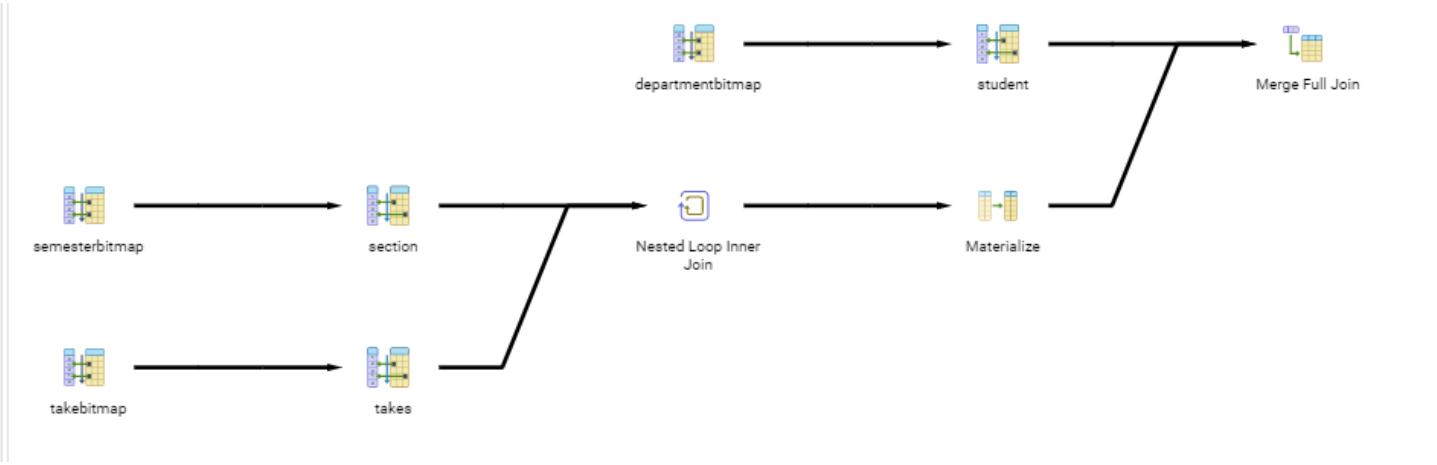
Cost was found to be equal:708.08 with an average execution time after running the query 3 times equal: (17.675+ 18.184+ 12.453)/3= 16.104 ms.

With bitmap:

Flags: set enable_seqscan=off;

Explain analyze output:

	QUERY PLAN text
1	Merge Full Join (cost=26.13..591.44 rows=33433 width=64) (actual time=0.089..3.292 rows=16467 loops=1)
2	-> Bitmap Heap Scan on student (cost=11.87..73.10 rows=499 width=24) (actual time=0.055..0.089 rows=499 loops=1)
3	Recheck Cond: ((department)::text = 'CS1'::text)
4	Heap Blocks: exact=4
5	-> Bitmap Index Scan on departmentbitmap (cost=0.00..11.74 rows=499 width=0) (actual time=0.048..0.048 rows=499 loops=1)
6	Index Cond: ((department)::text = 'CS1'::text)
7	-> Materialize (cost=14.27..100.59 rows=67 width=40) (actual time=0.030..0.649 rows=15969 loops=1)
8	-> Nested Loop (cost=14.27..100.42 rows=67 width=40) (actual time=0.026..0.067 rows=33 loops=1)
9	-> Bitmap Heap Scan on section s (cost=8.01..12.02 rows=2 width=28) (actual time=0.004..0.005 rows=1 loops=1)
10	Recheck Cond: (semester = 1)
11	Filter: (year = 2019)
12	Heap Blocks: exact=1
13	-> Bitmap Index Scan on semesterbitmap (cost=0.00..8.01 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=1)
14	Index Cond: (semester = 1)
15	-> Bitmap Heap Scan on takes t (cost=6.26..43.87 rows=33 width=12) (actual time=0.016..0.051 rows=33 loops=1)
16	Recheck Cond: (section_id = s.section_id)
17	Heap Blocks: exact=33
18	-> Bitmap Index Scan on takebitmap (cost=0.00..6.25 rows=33 width=0) (actual time=0.010..0.010 rows=33 loops=1)
19	Index Cond: (section_id = s.section_id)
20	Planning Time: 1.116 ms
21	Execution Time: 3.749 ms



"Merge Full Join (cost=26.13..591.44 rows=33433 width=64) (actual time=0.172..7.607 rows=16467 loops=1)"

"Execution Time: 3.749ms"

"Execution Time: 8.488 ms"

"Execution Time: 4.390 ms"

Explanation:

As we notice from query plan and explain analyze output that bitmap index is used

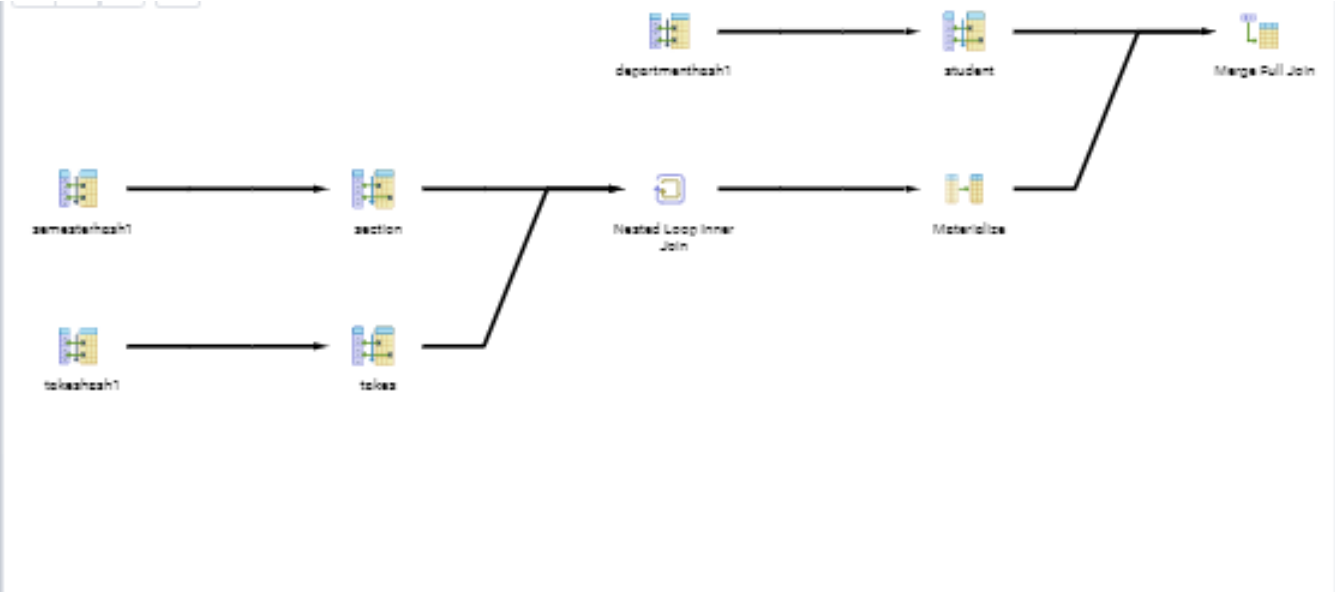
Cost was found to be equal: 591.44 with an average execution time after running the query 3 times equal: (3.749 + 8.488 + 8.488)/3= 6.908 ms. So, we notice that cost has decreased compared to without index case due to using bitmap index, as it makes the operation faster so the execution time also decreased to 6.908 ms so this execution time is accepted.

With hash:

Flags: set enable_seqscan=off;

Explain analyze output:

	QUERY PLAN text
1	Merge Full Join (cost=28.14..592.29 rows=33500 width=64) (actual time=0.081..6.232 rows=16500 loops=1)
2	-> Bitmap Heap Scan on student (cost=19.88..81.13 rows=500 width=24) (actual time=0.040..0.199 rows=500 loops=1)
3	Recheck Cond: ((department)::text = 'CS3'::text)
4	Heap Blocks: exact=5
5	-> Bitmap Index Scan on departmenthash1 (cost=0.00..19.75 rows=500 width=0) (actual time=0.030..0.030 rows=500 loops=1)
6	Index Cond: ((department)::text = 'CS3'::text)
7	-> Materialize (cost=8.26..92.59 rows=67 width=40) (actual time=0.039..1.009 rows=16001 loops=1)
8	-> Nested Loop (cost=8.26..92.41 rows=67 width=40) (actual time=0.036..0.075 rows=33 loops=1)
9	-> Bitmap Heap Scan on section s (cost=4.01..8.02 rows=2 width=28) (actual time=0.006..0.006 rows=1 loops=1)
10	Recheck Cond: (semester = 1)
11	Filter: (year = 2019)
12	Heap Blocks: exact=1
13	-> Bitmap Index Scan on semesterhash1 (cost=0.00..4.01 rows=1 width=0) (actual time=0.004..0.005 rows=1 loops=1)
14	Index Cond: (semester = 1)
15	-> Bitmap Heap Scan on takes t (cost=4.26..41.87 rows=33 width=12) (actual time=0.022..0.053 rows=33 loops=1)
16	Recheck Cond: (section_id = s.section_id)
17	Heap Blocks: exact=33
18	-> Bitmap Index Scan on takeshash1 (cost=0.00..4.25 rows=33 width=0) (actual time=0.016..0.016 rows=33 loops=1)
19	Index Cond: (section_id = s.section_id)
20	Planning Time: 1.472 ms
21	Execution Time: 6.874 ms



"Merge Full Join (cost=28.14..592.29 rows=33500 width=64) (actual time=0.081..6.232 rows=16500 loops=1)"

"Execution Time: 6.874 ms"

"Execution Time: 13.095 ms"

"Execution Time: 16.100 ms"

Explanation:

As we notice from query plan and explain analyze output that hash index is used

Cost was found to be equal: 592.29 with an average execution time after running the query 3 times equal: (6.874 + 13.095 + 16.100)/3= 12.023 ms. So, we notice that cost has decreased compared to without index case due to using hash-based index, as hash complexity is O(1), while without index is O(n) so the execution time also decreased to 12.023 ms so this execution time is accepted.

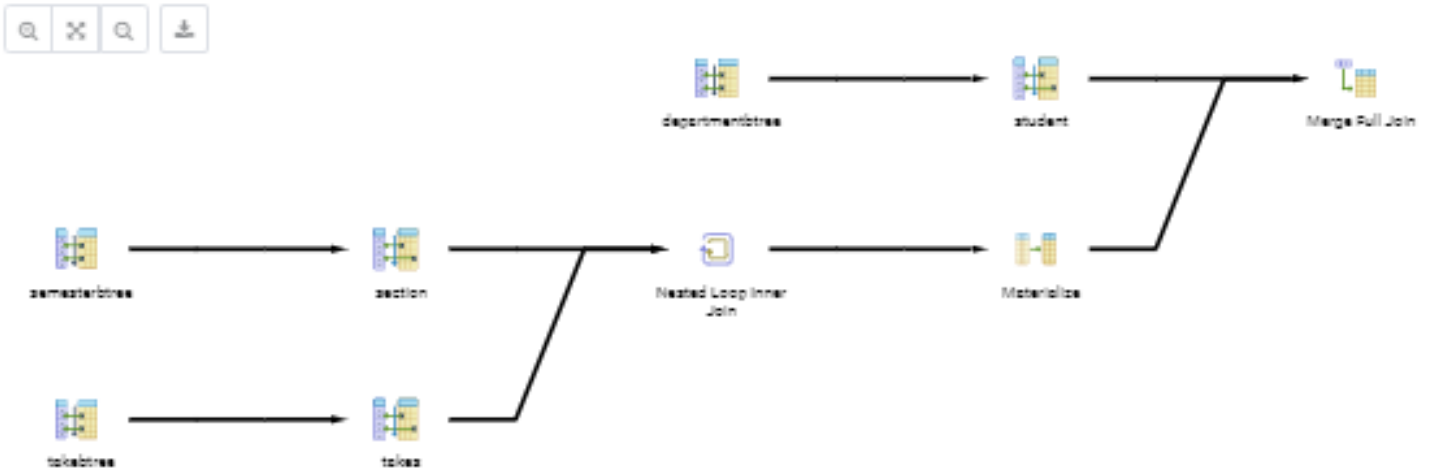
	QUERY PLAN text
1	Merge Full Join (cost=29.14..592.29 rows=33500 width=64) (actual time=0.091..6.232 rows=16500 loops=1)
2	-> Bitmap Heap Scan on student (cost=19.88..81.13 rows=500 width=24) (actual time=0.040..0.199 rows=500 loops=1)
3	Recheck Cond: ((department)::text = 'CS3'::text)
4	Heap Blocks: exact=5
5	-> Bitmap Index Scan on departmenthash1 (cost=0.00..19.75 rows=500 width=0) (actual time=0.030..0.030 rows=500 loops=1)
6	Index Cond: ((department)::text = 'CS3'::text)
7	-> Materialize (cost=8.26..92.58 rows=67 width=40) (actual time=0.039..1.009 rows=16001 loops=1)
8	-> Nested Loop (cost=8.26..92.41 rows=67 width=40) (actual time=0.036..0.075 rows=33 loops=1)
9	-> Bitmap Heap Scan on section s (cost=4.01..8.02 rows=2 width=28) (actual time=0.006..0.006 rows=1 loops=1)
10	Recheck Cond: (semester = 1)
11	Filter: (year = 2019)
12	Heap Blocks: exact=1
13	-> Bitmap Index Scan on semesterhash1 (cost=0.00..4.01 rows=1 width=0) (actual time=0.004..0.005 rows=1 loops=1)
14	Index Cond: (semester = 1)
15	-> Bitmap Heap Scan on takes t (cost=4.26..41.87 rows=33 width=12) (actual time=0.022..0.053 rows=33 loops=1)
16	Recheck Cond: (section_id = s.section_id)
17	Heap Blocks: exact=33
18	-> Bitmap Index Scan on takeshash1 (cost=0.00..4.25 rows=33 width=0) (actual time=0.016..0.016 rows=33 loops=1)
19	Index Cond: (section_id = s.section_id)
20	Planning Time: 1.472 ms
21	Execution Time: 6.874 ms

With btree:

Flags: set enable_seqscan=off;

Explain analyze output:

	text
1	Merge Full Join (cost=20.84..584.42 rows=33433 width=64) (actual time=0.210..9.832 rows=16467 loops=1)
2	-> Bitmap Heap Scan on student (cost=12.15..73.39 rows=499 width=24) (actual time=0.132..0.300 rows=499 loops=1)
3	Recheck Cond: ((department)::text = 'CS1'::text)
4	Heap Blocks: exact=4
5	-> Bitmap Index Scan on departmentbtree (cost=0.00..12.03 rows=499 width=0) (actual time=0.110..0.110 rows=499 loops=1)
6	Index Cond: ((department)::text = 'CS1'::text)
7	-> Materialize (cost=8.69..93.29 rows=67 width=40) (actual time=0.074..1.655 rows=15969 loops=1)
8	-> Nested Loop (cost=8.69..93.12 rows=67 width=40) (actual time=0.064..0.159 rows=33 loops=1)
9	-> Bitmap Heap Scan on section s (cost=4.15..8.17 rows=2 width=28) (actual time=0.010..0.010 rows=1 loops=1)
10	Recheck Cond: (semester = 1)
11	Filter: (year = 2019)
12	Heap Blocks: exact=1
13	-> Bitmap Index Scan on semesterbtree (cost=0.00..4.15 rows=1 width=0) (actual time=0.005..0.006 rows=1 loops=1)
14	Index Cond: (semester = 1)
15	-> Bitmap Heap Scan on takes t (cost=4.54..42.15 rows=33 width=12) (actual time=0.036..0.113 rows=33 loops=1)
16	Recheck Cond: (section_id = s.section_id)
17	Heap Blocks: exact=33
18	-> Bitmap Index Scan on takebtree (cost=0.00..4.53 rows=33 width=0) (actual time=0.026..0.026 rows=33 loops=1)
19	Index Cond: (section_id = s.section_id)
20	Planning Time: 0.956 ms
21	Execution Time: 10.790 ms



"Merge Full Join (cost=20.84..584.42 rows=33433 width=64) (actual time=0.210..9.832 rows=16467 loops=1)"

"Execution Time: 10.790 ms"

"Execution Time: 9.011 ms"

"Execution Time: 28.726 ms"

Explanation:

As we notice from query plan and explain analyze output that btree index is used

Cost was found to be equal:584.42 with an average execution time after running the query 3 times equal: (10.790 + 9.011 + 28.726)/3= 16.175 ms. so, we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to16.175 ms so this execution time is accepted.

The best scenario:

	Without index	btree	hash	bitmap
Cost	708.08	584.42	592.29	591.44
Exec.	16.104	16.175	12.023	6.908

So,btree index is the best index for this query as it’s the least cost.

Schema 2

Modifications on Data Insertions:

As required in the project description, that the number of Employees should be 5000, 30 departments, 20 department locations, and 600 projects.

So in order for the results of the query to be not empty, I have changed the following data population code:

1- For the populateEmployee in order to make sure that the 5000 Employee is inserted I've changed the for loop condition for the I to be less than 5001 " for(int i =1 ; i < 5001 ; I++) ".

2- Also in order to satisfy that there are salaries greater than or equal to 40000 I made an int variable called salary which is a random variable (I.e. int salary = new Random().nextInt(40000+1)+10000;).

3- I've also changed the if condition in line 270 to be if(i> 500) then result = "F";

4- I've also changed the variable "i" in if condition line 272 that is inserted in dno to be i%30+1 as to make sure of constraints.

5- For the populateDepartment in order to make sure that the departments of total 30, I've changed the for loop to be (I.e. for (int i =1 ; i< 31 ; i++)).

6- And the if condition in line 288, I've changed the value of variable i in MgrSSN to be : i%5000+1, in order to make sure that it respect the constraints of foreign key.

7- For the populateDeptLocations same as before I changed the i value in for loop to be less than 21 to make sure that I have 20 department locations as stated in the requirements, And I've also changed the if condition in line 297 the value of I in Dnumber to be (i%30+1), in order to make sure that it respect the constraints of foreign key.

8- For the populateProject same as before I changed the i value in for loop to be less than 601 to make sure that I have 600 department locations as stated in the requirements, And the if condition in line 307, I've changed the "i" value to be i%30+1 in Dnumber, in order to make sure that it respect the constraints of foreign key.

9- For the populateDependent I haven't changed the for loop, it have the same value, however in line 329 I've changed the "I" value in Essn to be I%5000+1, in order to make sure that it respect the constraints of foreign key.

10- For the PopulateWorksOn I haven't changed the for loop, it have the same value, however in line 316 I've changed the "i" value in Essn to be i%5000+1, and in pNo to be i%600+1, in order to make sure that it respect the constraints of foreign key.

Query 2:

Output of Query : 201

Without an index:

Flags :

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = off;

set enable_indexonlyscan = off;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = off;

set enable_bitmapscan= off;

set enable_sort = on;

set enable_tidscan = off;

	QUERY PLAN text
1	Unique (cost=30000000629.57..30000000631.82 rows=450 width=4) (actual time=6.331..6.332 rows=1 loops=1)
2	-> Sort (cost=30000000629.57..30000000630.69 rows=450 width=4) (actual time=6.327..6.328 rows=1 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 25kB
5	-> Seq Scan on project (cost=30000000591.74..30000000609.74 rows=450 width=4) (actual time=6.140..6.313 rows=1 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	Rows Removed by Filter: 599
8	SubPlan 1
9	-> Nested Loop (cost=20000000000.00..20000000167.25 rows=1 width=4) (actual time=1.599..1.599 rows=0 loops=1)
10	-> Nested Loop (cost=10000000000.00..10000000146.25 rows=1 width=0) (actual time=1.598..1.598 rows=0 loops=1)
11	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
12	Rows Removed by Join Filter: 30
13	-> Seq Scan on employee e (cost=0.00..144.50 rows=1 width=8) (actual time=0.018..1.571 rows=1 loops=1)
14	Filter: (lname = 'employee1'::bpchar)
15	Rows Removed by Filter: 4999
16	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.014..0.018 rows=30 loops=1)
17	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
18	SubPlan 2
19	-> Nested Loop (cost=10000000000.00..10000000424.48 rows=2 width=4) (actual time=1.565..4.408 rows=1 loops=1)
20	Join Filter: (works_on.essn = employee.ssn)
21	Rows Removed by Join Filter: 9998
22	-> Seq Scan on employee (cost=0.00..144.50 rows=1 width=4) (actual time=0.057..1.491 rows=1 loops=1)
23	Filter: (lname = 'employee1'::bpchar)
24	Rows Removed by Filter: 4999
25	-> Seq Scan on works_on (cost=0.00..154.99 rows=9999 width=8) (actual time=0.018..1.448 rows=9999 loops=1)
26	Planning Time: 0.507 ms
27	Execution Time: 6.418 ms

Estimated cost:

.30000000609.74

1st run execution time: 6.418 ms

2nd run execution time: 7.636 ms

3rd run execution time : 5.685 ms

Average execution time = 6.579 ms

With an index:

B+ Tree Index :

Flags :

set enable_seqscan=off;

set enable_indexscan=on;

create index be on Employee using btree(Dno);

create index bp on Project using btree(Pnumber);

	QUERY PLAN text
1	Unique (cost=323.81..364.93 rows=450 width=4) (actual time=4.459..4.821 rows=1 loops=1)
2	-> Index Only Scan using bp on project (cost=323.81..363.81 rows=450 width=4) (actual time=4.455..4.816 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=0.00..167.25 rows=1 width=4) (actual time=2.020..2.020 rows=0 loops=1)
8	-> Nested Loop (cost=0.00..146.25 rows=1 width=0) (actual time=2.018..2.019 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Seq Scan on employee e (cost=0.00..144.50 rows=1 width=8) (actual time=0.034..1.987 rows=1 loops=1)
12	Filter: (lname = 'employee1'::bpchar)
13	Rows Removed by Filter: 4999
14	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.018..0.023 rows=30 loops=1)
15	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
16	SubPlan 2
17	-> Nested Loop (cost=0.29..156.28 rows=2 width=4) (actual time=0.057..2.179 rows=1 loops=1)
18	-> Seq Scan on employee (cost=0.00..144.50 rows=1 width=4) (actual time=0.024..2.143 rows=1 loops=1)
19	Filter: (lname = 'employee1'::bpchar)
20	Rows Removed by Filter: 4999
21	-> Index Only Scan using works_on_pkey on works_on (cost=0.29..11.76 rows=2 width=8) (actual time=0.029..0.031 rows=1 loops=1)
22	Index Cond: (essn = employee.ssn)

23	Heap Fetches: 1
24	Planning Time: 1.332 ms
25	Execution Time: 4.953 ms

Estimated cost:

364.93

1st run execution time: 4.953 ms

2nd run execution time: 7.357 ms

3rd run execution time : 8.942 ms

Average execution time = 7.084 ms

After creating btree index on last name in table Employee with the previous indexes the Execution time reduced :

create index bel on Employee using btree(Lname);

1st run execution time: 0.457 ms

2nd run execution time:0.879 ms

3rd run execution time : 0.726 ms

Average execution time = 0.687 ms

	QUERY PLAN text
1	Unique (cost=51.41..92.53 rows=450 width=4) (actual time=0.240..0.375 rows=1 loops=1)
2	-> Index Only Scan using bp on project (cost=51.41..91.41 rows=450 width=4) (actual time=0.239..0.374 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=0.28..31.05 rows=1 width=4) (actual time=0.121..0.121 rows=0 loops=1)
8	-> Nested Loop (cost=0.28..10.05 rows=1 width=0) (actual time=0.121..0.121 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Index Scan using bel on employee e (cost=0.28..8.30 rows=1 width=8) (actual time=0.100..0.101 rows=1 loops=1)
12	Index Cond: (lname = 'employee1'::bpchar)
13	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.012..0.014 rows=30 loops=1)
14	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
15	SubPlan 2
16	-> Nested Loop (cost=0.57..20.08 rows=2 width=4) (actual time=0.021..0.022 rows=1 loops=1)
17	-> Index Scan using bel on employee (cost=0.28..8.30 rows=1 width=4) (actual time=0.008..0.008 rows=1 loops=1)
18	Index Cond: (lname = 'employee1'::bpchar)
19	-> Index Only Scan using works_on_pkey on works_on (cost=0.29..11.76 rows=2 width=8) (actual time=0.012..0.012 rows=1 loops=1)
20	Index Cond: (essn = employee.ssn)
21	Heap Fetches: 1
22	Planning Time: 1.995 ms
23	Execution Time: 0.457 ms

After creating an index on Essn in Table Works_on

Estimated cost:

create index bessn on Works_on using btree(Essn);

	QUERY PLAN text
1	Unique (cost=50.97..92.10 rows=450 width=4) (actual time=0.251..0.434 rows=1 loops=1)
2	-> Index Only Scan using bp on project (cost=50.97..90.97 rows=450 width=4) (actual time=0.250..0.433 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=0.28..31.05 rows=1 width=4) (actual time=0.061..0.061 rows=0 loops=1)
8	-> Nested Loop (cost=0.28..10.05 rows=1 width=0) (actual time=0.060..0.061 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Index Scan using bel on employee e (cost=0.28..8.30 rows=1 width=8) (actual time=0.032..0.033 rows=1 loops=1)
12	Index Cond: (lname = 'employee1'::bpchar)
13	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.017..0.020 rows=30 loops=1)
14	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
15	SubPlan 2
16	-> Nested Loop (cost=0.57..19.64 rows=2 width=4) (actual time=0.055..0.057 rows=1 loops=1)
17	-> Index Scan using bel on employee (cost=0.28..8.30 rows=1 width=4) (actual time=0.010..0.010 rows=1 loops=1)
18	Index Cond: (lname = 'employee1'::bpchar)
19	-> Index Scan using bessn on works_on (cost=0.29..11.32 rows=2 width=8) (actual time=0.044..0.044 rows=1 loops=1)
20	Index Cond: (essn = employee.ssn)
21	Planning Time: 3.188 ms
22	Execution Time: 0.550 ms

Estimated cost:

92.10

1st run execution time: 0.550 ms

2nd run execution time:0.435 ms

3rd run execution time : 0.350 ms

Average execution time = 0.445 ms

Explanation :

The cost has decreased due to the usage of the B-tree index that has time complexity of O(log n) instead of the linear searching of O (n). And the execution time has decreased as well.

drop index be;

drop index bp;

drop index bel;

drop index bessn;

Bitmap Index :

Flags:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= on;

set enable_sort = off;

create extension btree_gin;

create index bmE on Employee using gin(ssn,dno);

create index bmW on Works_on using gin(Essn);

create index bml on Employee using gin(Lname);

create index Pnobitmap on Project using gin(Pnumber);

	QUERY PLAN text
1	Unique (cost=66.27..107.40 rows=450 width=4) (actual time=0.424..0.781 rows=1 loops=1)
2	-> Index Only Scan using project_pkey on project (cost=66.27..106.27 rows=450 width=4) (actual time=0.422..0.778 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=12.01..38.77 rows=1 width=4) (actual time=0.097..0.097 rows=0 loops=1)
8	-> Nested Loop (cost=12.01..17.77 rows=1 width=0) (actual time=0.096..0.096 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Bitmap Heap Scan on employee e (cost=12.01..16.02 rows=1 width=8) (actual time=0.051..0.052 rows=1 loops=1)
12	Recheck Cond: (lname = 'employee1'::bpchar)
13	Heap Blocks: exact=1
14	-> Bitmap Index Scan on bml (cost=0.00..12.01 rows=1 width=0) (actual time=0.045..0.045 rows=1 loops=1)
15	Index Cond: (lname = 'employee1'::bpchar)
16	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.024..0.030 rows=30 loops=1)
17	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
18	SubPlan 2
19	-> Nested Loop (cost=16.31..27.22 rows=2 width=4) (actual time=0.059..0.061 rows=1 loops=1)
20	-> Bitmap Heap Scan on employee (cost=12.01..16.02 rows=1 width=4) (actual time=0.023..0.023 rows=1 loops=1)
21	Recheck Cond: (lname = 'employee1'::bpchar)
22	Heap Blocks: exact=1
23	-> Bitmap Index Scan on bml (cost=0.00..12.01 rows=1 width=0) (actual time=0.021..0.021 rows=1 loops=1)
24	Index Cond: (lname = 'employee1'::bpchar)
25	-> Bitmap Heap Scan on works_on (cost=4.30..11.18 rows=2 width=8) (actual time=0.017..0.017 rows=1 loops=1)
26	Recheck Cond: (essn = employee.ssn)
27	Heap Blocks: exact=1
28	-> Bitmap Index Scan on works_on_pkey (cost=0.00..4.30 rows=2 width=0) (actual time=0.012..0.012 rows=1 loops=1)
29	Index Cond: (essn = employee.ssn)
30	Planning Time: 1.360 ms
31	Execution Time: 1.010 ms

Estimated cost:

.107.40

1st run execution time: 1.010 ms

2nd run execution time: 0.515 ms

3rd run execution time : 0.337 ms

Average execution time = 0.287 ms

Explanation: The cost has increased but the execution time has decreased because it uses bit-wise operations to filter the columns, which is really fast.

drop index bmE;

drop index bml;

drop index bmW;

drop index Pnobitmap;

Hash index:

Flags:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

create index PHash on Employee using hash(Lname);

create index EHash on Employee using hash(ssn);

create index dHash on Employee using hash(dno);

	QUERY PLAN text
1	Unique (cost=50.84..91.97 rows=450 width=4) (actual time=0.736..1.102 rows=1 loops=1)
2	-> Index Only Scan using project_pkey on project (cost=50.84..90.84 rows=450 width=4) (actual time=0.734..1.100 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=0.00..30.77 rows=1 width=4) (actual time=0.091..0.091 rows=0 loops=1)
8	-> Nested Loop (cost=0.00..9.77 rows=1 width=0) (actual time=0.091..0.091 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Index Scan using phash on employee e (cost=0.00..8.02 rows=1 width=8) (actual time=0.023..0.026 rows=1 loops=1)
12	Index Cond: (lname = 'employee1'::bpchar)
13	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.035..0.040 rows=30 loops=1)
14	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
15	SubPlan 2
16	-> Nested Loop (cost=0.29..19.80 rows=2 width=4) (actual time=0.179..0.184 rows=1 loops=1)
17	-> Index Scan using phash on employee (cost=0.00..8.02 rows=1 width=4) (actual time=0.005..0.007 rows=1 loops=1)
18	Index Cond: (lname = 'employee1'::bpchar)
19	-> Index Only Scan using works_on_pkey on works_on (cost=0.29..11.76 rows=2 width=8) (actual time=0.169..0.171 rows=1 loops=1)
20	Index Cond: (essn = employee.ssn)
21	Heap Fetches: 1
21	Heap Fetches: 1
22	Planning Time: 7.642 ms
23	Execution Time: 1.269 ms

Estimated cost:

91.97

1st run: 1.269

2nd run: 0.829

3rd run: 0.314

Average execution time : 0.804

create index dehash on Department using hash(Mgr_snn);

create index Pnohash on Project using hash(Pnumber);

	QUERY PLAN text
1	Unique (cost=50.84..91.97 rows=450 width=4) (actual time=0.235..0.516 rows=1 loops=1)
2	-> Index Only Scan using project_pkey on project (cost=50.84..90.84 rows=450 width=4) (actual time=0.234..0.514 rows=1 loops=1)
3	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
4	Rows Removed by Filter: 599
5	Heap Fetches: 600
6	SubPlan 1
7	-> Nested Loop (cost=0.00..30.77 rows=1 width=4) (actual time=0.050..0.051 rows=0 loops=1)
8	-> Nested Loop (cost=0.00..9.77 rows=1 width=0) (actual time=0.050..0.050 rows=0 loops=1)
9	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
10	Rows Removed by Join Filter: 30
11	-> Index Scan using phash on employee e (cost=0.00..8.02 rows=1 width=8) (actual time=0.014..0.015 rows=1 loops=1)
12	Index Cond: (lname = 'employee1'::bpchar)
13	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.022..0.026 rows=30 loops=1)
14	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (never executed)
15	SubPlan 2
16	-> Nested Loop (cost=0.29..19.80 rows=2 width=4) (actual time=0.019..0.022 rows=1 loops=1)
17	-> Index Scan using phash on employee (cost=0.00..8.02 rows=1 width=4) (actual time=0.003..0.004 rows=1 loops=1)
18	Index Cond: (lname = 'employee1'::bpchar)
19	-> Index Only Scan using works_on_pkey on works_on (cost=0.29..11.76 rows=2 width=8) (actual time=0.014..0.015 rows=1 loops=1)
20	Index Cond: (essn = employee.ssn)
21	Heap Fetches: 1
22	Planning Time: 0.951 ms

21	Heap Fetches: 1
22	Planning Time: 0.951 ms
23	Execution Time: 0.609 ms

Estimated cost:

91.97

1st run execution time: 1.269 ms

2nd run execution time: 0.694 ms

3rd run execution time : 0.431 ms

Average execution time = 0.798 ms

Explanation: The cost has decreased since we used hash-based index.

```
drop index PHash;
drop index EHash;
drop index dHash;
drop index dehash;
drop index Pnohash;
```

Best Scenario for Query 2:

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using Bitmap index, since it has the least execution time

---

Query 3:

Without index:

Flags:

set enable_seqscan = off;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = off;

set enable_indexonlyscan = off;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..362438.25 rows=2500 width=42) (actual time=255.025..356.635 rows=1 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4999
4	SubPlan 1
5	-> Seq Scan on employee employee_1 (cost=0.00..144.50 rows=167 width=4) (actual time=0.005..0.055 rows=5 loops=5000)
6	Filter: (dno = 5)
7	Rows Removed by Filter: 114
8	Planning Time: 0.430 ms
9	Execution Time: 356.714 ms

Estimated cost:

362438.25

1st run : execution time : 140.324 ms

2nd run : execution time : 124.582 ms

3rd run : execution time ; 126.978 ms

Average execution time : 130.628ms

With Index:

B+ tree index:

Flags:

set enable_seqscan = off;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

create index bmE on Employee using btree(lname,fname,salary,dno);

	QUERY PLAN text
1	Index Only Scan using bme on employee (cost=0.28..1359968.30 rows=2500 width=42) (actual time=104.243..148.227 rows=1 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4999
4	Heap Fetches: 5000
5	SubPlan 1
6	-> Index Only Scan using bme on employee employee_1 (cost=0.28..543.05 rows=167 width=4) (actual time=0.012..0.027 rows=5 loops=5000)
7	Index Cond: (dno = 5)
8	Heap Fetches: 24925
9	Planning Time: 4.798 ms
10	Execution Time: 148.297 ms

Estimated time:

1359968.30

1st run : execution time : 148.297 ms

2nd run : execution time : 74.242 ms

3rd run : execution time : 78.617 ms

Average execution time : 100.385 ms

Drop index bmE;

Explanation: The cost has decreased due to the usage of the B+ tree index that has time complexity of O (log n) instead of the linear searching of O (n), and we can see that it affects in the execution time.

With Bitmap index:

Flags:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= on;

set enable_sort = off;

set enable_tidscan = off;

create extension btree_gin;

create index bmE on Employee using gin(lname,fname,salary,dno);

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..277878.25 rows=2500 width=42) (actual time=150.728..208.426 rows=1 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4999
4	SubPlan 1
5	-> Bitmap Heap Scan on employee employee_1 (cost=13.29..97.38 rows=167 width=4) (actual time=0.030..0.032 rows=5 loops=5000)
6	Recheck Cond: (dno = 5)
7	Heap Blocks: exact=14225
8	-> Bitmap Index Scan on bme (cost=0.00..13.25 rows=167 width=0) (actual time=0.025..0.025 rows=167 loops=5000)
9	Index Cond: (dno = 5)
10	Planning Time: 0.379 ms
11	Execution Time: 208.517 ms

Estimated cost:

277878.25

1st run : execution time : 208.517 ms

2nd run : execution time : 166.775 ms

3rd run : execution time : 156.318 ms

Average execution time : 177.203 ms

Explanation: The cost has decreased since we used bitmap index that uses bit-wise operations to filter the columns.

Hash index :

Flags:

set enable_seqscan = off;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

create index dnHash on Employee using hash(dno);

	QUERY PLAN text
1	Seq Scan on employee (cost=10000000000.00..10000848494.50 rows=2500 width=42) (actual time=14.562..20.345 rows=1 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4999
4	SubPlan 1
5	-> Index Scan using dnhash on employee employee_1 (cost=0.00..338.92 rows=167 width=4) (actual time=0.003..0.003 rows=2 loops=5000)
6	Index Cond: (dno = 5)
7	Planning Time: 1.304 ms
8	Execution Time: 20.393 ms

Estimated cost:

10000848494.50

1st Execution time : 20.393 ms

2nd Execution time : 21.759 ms

3rd Execution time : 22.992 ms

Average Execution time : 21.714 ms

Explanation: The cost has increased since the seqScan flag is off however the time complexity is reduced due to the use of hash index which has a time complexity of O(1).

Best Scenario for Query 3:

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using Hash Index, since it has the least execution time.

Query 4:

Flags:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = off;

set enable_indexonlyscan = off;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = off;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

	QUERY PLAN text
1	Seq Scan on employee e (cost=0.00..666350.75 rows=2500 width=42) (actual time=27.970..6536.580 rows=5000 loops=1)
2	Filter: (SubPlan 1)
3	SubPlan 1
4	-> Seq Scan on dependent d (cost=0.00..253.98 rows=4999 width=4) (actual time=0.680..1.170 rows=2501 loops=5000)
5	Filter: ((e.fname <> dependent_name) AND (e.sex <> sex))
6	Rows Removed by Filter: 4500
7	Planning Time: 5.909 ms
8	Execution Time: 6537.153 ms

Estimated cost:

.666350.75

1st Execution time : 6537.153 ms

2nd Execution time : 6356.290 ms

3rd Execution time : 6339.351 ms

Average Execution time : 6410.931 ms

With Index :

Explanation:

The behavior of the "!= " in the query makes it almost impossible to create an index on it, as it will always chooses to do sequential scan and sets the boolean seqScan on automatically although even if it's set off.

Best Scenario for Query 4:

Referring to the previous Explanation the best scenario is without using an index,

As the Query does Linear Scan of O(n) complexity due to the "!=" operator.

Query 5:

Without Index:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = off;

set enable_indexonlyscan = off;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = off;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

	QUERY PLAN text
1	Nested Loop Semi Join (cost=10000000000.00..10001096864.77 rows=5000 width=42) (actual time=0.976..1529.625 rows=5000 loops=...
2	Join Filter: (employee.ssn = dependent.essn)
3	Rows Removed by Join Filter: 12497500
4	-> Seq Scan on employee (cost=0.00..132.00 rows=5000 width=46) (actual time=0.018..0.715 rows=5000 loops=1)
5	-> Seq Scan on dependent (cost=0.00..203.99 rows=9999 width=4) (actual time=0.002..0.150 rows=2501 loops=5000)
6	Planning Time: 0.179 ms
7	Execution Time: 1529.896 ms

Estimated cost:

.10001096864.77

1st Execution time : 1529.896 ms

2nd Execution time : 1593.303 ms ms

3rd Execution time : 1562.468 ms

Average Execution time : 1561.889 ms

With Index :

B+ tree index :

Flags:

set enable_seqscan = off;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = off;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

create index bte on Employee using btree(ssn);

	QUERY PLAN text
1	Nested Loop Semi Join (cost=10000000000.57..10000002358.00 rows=5000 width=42) (actual time=0.099..9.900 rows=5000 loops=1)
2	-> Index Scan using bte on employee (cost=0.28..224.28 rows=5000 width=46) (actual time=0.031..1.317 rows=5000 loops=1)
3	-> Index Only Scan using dependent_pkey on dependent (cost=0.29..0.48 rows=2 width=4) (actual time=0.001..0.001 rows=1 loops=5000)
4	Index Cond: (essn = employee.ssn)
5	Heap Fetches: 5000
6	Planning Time: 1.674 ms
7	Execution Time: 10.077 ms

Estimated Cost:

10000002358.00

1st Exec time : 10.077 ms

2nd exec time : 10.276 ms

3rd exec time : 14.971 ms

Average Execution time : 11.774 ms

Explanation: B-Tree has a time complexity of O (log n), and the column has no duplicates which makes the operation faster and we can notice that the cost has slightly decreased and the execution time also has decreased a lot more than that without index.

drop index bte;

Bitmap index :

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = on;

set enable_indexonlyscan = on;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = on;

set enable_bitmapscan= on;

set enable_sort = off;

create index bmd on dependent using gin(essn);

create index bmE on Employee using gin(ssn,fname);

drop index bmE;

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.57..2358.00 rows=5000 width=42) (actual time=0.057..28.114 rows=5000 loops=1)
2	-> Index Scan using employee_pkey on employee (cost=0.28..224.28 rows=5000 width=46) (actual time=0.031..2.898 rows=5000 loops=1)
3	-> Index Only Scan using dependent_pkey on dependent (cost=0.29..0.48 rows=2 width=4) (actual time=0.004..0.004 rows=1 loops=5000)
4	Index Cond: (essn = employee.ssn)
5	Heap Fetches: 5000
6	Planning Time: 0.518 ms
7	Execution Time: 28.663 ms

Estimated Cost:

2358.00

1st Execution Time: 28.663 ms

2nd Execution Time: 35.427 ms

3rd Execution Time: 15.095 ms

Average Execution time: 26.395ms

Explanation: The difference is that applying the bitmap index makes the Query faster the cost has reduced a lot and also the execution time.

Hash index :

```
set enable_seqscan = off;
set enable_gathermerge= off;
set enable_hashagg = off;
set enable_hashjoin = off;
set enable_indexscan = on;
set enable_indexonlyscan = on;
set enable_material = off;
set enable_mergejoin = off;
set enable_nestloop = on;
set enable_bitmapscan= off;
set enable_sort = off;
set enable_tidscan = off;
```

```
create index dhash on dependent using hash(essn);
```

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.28..860.99 rows=5000 width=42) (actual time=0.057..18.777 rows=5000 loops=1)
2	-> Index Scan using employee_pkey on employee (cost=0.28..224.28 rows=5000 width=46) (actual time=0.034..2.510 rows=5000 loops=1)
3	-> Index Scan using dhash on dependent (cost=0.00..0.17 rows=2 width=4) (actual time=0.002..0.002 rows=1 loops=5000)
4	Index Cond: (essn = employee.ssn)
5	Planning Time: 0.414 ms
6	Execution Time: 19.242 ms

Estimated Cost:

860.99

1st Execution Time: 19.242 ms

2nd Execution Time: 7.574 ms

3rd Execution Time: 15.767 ms

Average Execution time: 14.194 ms

drop index dhash;

Explanation: The hash-based index , it gives a better performance than without applying any indices, especially with having no duplicates on the column and also it appears in the cost that has reduced and also the execution time.

Best Scenario for Query 5:

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using B-Tree index, since it has the least execution time.

Query 6:

Without index:

set enable_seqscan = on;

set enable_gathermerge= off;

set enable_hashagg = off;

set enable_hashjoin = off;

set enable_indexscan = off;

set enable_indexonlyscan = off;

set enable_material = off;

set enable_mergejoin = off;

set enable_nestloop = off;

set enable_bitmapscan= off;

set enable_sort = off;

set enable_tidscan = off;

	QUERY PLAN text
1	GroupAggregate (cost=30000000439.19..30000002479.18 rows=9 width=12) (actual time=3.087..29.940 rows=30 loops=1)
2	Group Key: department.dnumber
3	-> Nested Loop (cost=30000000439.19..30000002479.04 rows=9 width=4) (actual time=2.084..29.767 rows=1263 loops=1)
4	Join Filter: (department.dnumber = employee.dno)
5	Rows Removed by Join Filter: 36627
6	-> Nested Loop (cost=20000000439.19..20000000876.42 rows=10 width=8) (actual time=2.044..3.428 rows=30 loops=1)
7	Join Filter: (department.dnumber = employee_1.dno)
8	Rows Removed by Join Filter: 435
9	-> GroupAggregate (cost=10000000439.19..10000000477.07 rows=10 width=4) (actual time=2.027..3.049 rows=30 loops=1)
10	Group Key: employee_1.dno
11	Filter: (count(*) > 2)
12	-> Sort (cost=10000000439.19..10000000451.69 rows=5000 width=4) (actual time=1.983..2.493 rows=5000 loops=1)
13	Sort Key: employee_1.dno
14	Sort Method: quicksort Memory: 427kB
15	-> Seq Scan on employee employee_1 (cost=0.00..132.00 rows=5000 width=4) (actual time=0.028..0.988 rows=5000 loops=1)
16	-> Seq Scan on department (cost=0.00..23.30 rows=1330 width=4) (actual time=0.002..0.003 rows=16 loops=30)
17	-> Seq Scan on employee (cost=0.00..144.50 rows=1261 width=4) (actual time=0.002..0.773 rows=1263 loops=30)
18	Filter: (salary > 40000)
19	Rows Removed by Filter: 3737
20	Planning Time: 0.346 ms
21	Execution Time: 30.213 ms

Estimated Cost:

.30000002479.18

1st Execution Time: 30.213 ms

2nd Execution Time: 47.425 ms

3rd Execution Time: 47.810 ms

Average Execution time: 41.816 ms

With Index:

B+ Tree index:

```
set enable_seqscan = off;
set enable_gathermerge= off;
set enable_hashagg = off;
set enable_hashjoin = off;
set enable_indexscan = on;
set enable_indexonlyscan = on;
set enable_material = off;
set enable_mergejoin = off;
set enable_nestloop = on;
set enable_bitmapscan= off;
set enable_sort = off;
set enable_tidscan = off;

create index btE on Employee using btree(salary,dno);
create index btd on department using btree(dnumber);
```

	QUERY PLAN text
1	GroupAggregate (cost=10000000774.90..10000001096.47 rows=30 width=12) (actual time=11.029..19.226 rows=30 loops=1)
2	Group Key: department.dnumber
3	-> Nested Loop (cost=10000000774.90..10000001094.07 rows=420 width=4) (actual time=10.701..18.845 rows=1263 loops=1)
4	-> Nested Loop (cost=10000000774.61..10000000826.08 rows=10 width=8) (actual time=10.643..13.291 rows=30 loops=1)
5	-> GroupAggregate (cost=10000000774.48..10000000812.35 rows=10 width=4) (actual time=10.352..12.837 rows=30 loops=1)
6	Group Key: employee_1.dno
7	Filter: (count(*) > 2)
8	-> Sort (cost=10000000774.48..10000000786.98 rows=5000 width=4) (actual time=10.247..11.334 rows=5000 loops=1)
9	Sort Key: employee_1.dno
10	Sort Method: quicksort Memory: 427kB
11	-> Index Only Scan using bte on employee employee_1 (cost=0.28..467.28 rows=5000 width=4) (actual time=0.046..6.991 rows=5000 loops=1)
12	Heap Fetches: 5000
13	-> Index Only Scan using btd on department (cost=0.14..1.35 rows=1 width=4) (actual time=0.013..0.013 rows=1 loops=30)
14	Index Cond: (dnumber = employee_1.dno)
15	Heap Fetches: 30
16	-> Index Only Scan using bte on employee (cost=0.28..26.38 rows=42 width=4) (actual time=0.031..0.175 rows=42 loops=30)
17	Index Cond: ((salary > 40000) AND (dno = department.dnumber))
18	Heap Fetches: 1263
19	Planning Time: 4.845 ms
20	Execution Time: 19.722 ms

Estimated Cost:

10000001096.47

1st Execution Time: 19.722 ms

2nd Execution Time: 20.302 ms

3rd Execution Time: 15.178 ms

Average Execution time: 18.400 ms

drop index btE;

drop index btd;

Explanation: The reason of the difference before and after applying the index is that the B-Tree has a time complexity of O (log n), which makes the Query faster and we can notice that in the cost and the execution time that's reduced a lot.

Bitmap index:

```
set enable_seqscan = off;
set enable_gathermerge= off;
set enable_hashagg = off;
set enable_hashjoin = off;
set enable_indexscan = on;
set enable_indexonlyscan = on;
set enable_material = off;
set enable_mergejoin = off;
set enable_nestloop = on;
set enable_bitmapscan= on;
set enable_sort = off;
set enable_tidscan = off;

create index bmE on Employee using gin(salary,dno);
create index bmd on department using gin(dnumber);
```

	QUERY PLAN text
1	GroupAggregate (cost=40000000444.56..40000000658.79 rows=30 width=12) (actual time=3.083..10.112 rows=30 loops=1)
2	Group Key: department.dnumber
3	-> Nested Loop (cost=40000000444.56..40000000656.39 rows=420 width=4) (actual time=2.803..9.983 rows=1263 loops=1)
4	-> Nested Loop (cost=30000000439.33..30000000490.80 rows=10 width=8) (actual time=2.549..3.478 rows=30 loops=1)
5	-> GroupAggregate (cost=20000000439.19..20000000477.07 rows=10 width=4) (actual time=2.449..3.304 rows=30 loops=1)
6	Group Key: employee_1.dno
7	Filter: (count(*) > 2)
8	-> Sort (cost=20000000439.19..20000000451.69 rows=5000 width=4) (actual time=2.405..2.795 rows=5000 loops=1)
9	Sort Key: employee_1.dno
10	Sort Method: quicksort Memory: 427kB
11	-> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.012..1.318 rows=5000 loops=1)
12	-> Index Only Scan using department_pkey on department (cost=0.14..1.35 rows=1 width=4) (actual time=0.005..0.005 rows=1 loops=30)
13	Index Cond: (dnumber = employee_1.dno)
14	Heap Fetches: 30
15	-> Bitmap Heap Scan on employee (cost=5.23..16.14 rows=42 width=4) (actual time=0.185..0.206 rows=42 loops=30)
16	Recheck Cond: ((salary > 40000) AND (dno = department.dnumber))
17	Heap Blocks: exact=1104
18	-> Bitmap Index Scan on bme (cost=0.00..5.22 rows=42 width=0) (actual time=0.180..0.180 rows=42 loops=30)
19	Index Cond: ((salary > 40000) AND (dno = department.dnumber))
20	Planning Time: 2.625 ms
21	Execution Time: 10.300 ms

Estimated Cost:

40000000658.79

1st Execution Time: 10.300 ms

2nd Execution Time: 18.326 ms

3rd Execution Time: 19.594 ms

Average Execution Time: 16.073 ms

Explanation: The difference is that applying the bitmap index makes the Query fun faster and we can notice that in the execution time that has reduced slightly however due to turning the seqScan flag off the cost has increased.

Hash Index:

```
set enable_seqscan = off;
set enable_gathermerge= off;
set enable_hashagg = off;
set enable_hashjoin = off;
set enable_indexscan = on;
set enable_indexonlyscan = on;
set enable_material = off;
set enable_mergejoin = off;
set enable_nestloop = on;
set enable_bitmapscan= off;
set enable_sort = off;
set enable_tidscan = off;

create index ehash on Employee using hash(dno);
```

	QUERY PLAN text
1	GroupAggregate (cost=20000000439.33..20000000685.46 rows=30 width=12) (actual time=2.463..6.595 rows=30 loops=1)
2	Group Key: department.dnumber
3	-> Nested Loop (cost=20000000439.33..20000000683.06 rows=420 width=4) (actual time=2.302..6.434 rows=1263 loops=1)
4	-> Nested Loop (cost=20000000439.33..20000000490.80 rows=10 width=8) (actual time=2.290..3.368 rows=30 loops=1)
5	-> GroupAggregate (cost=20000000439.19..20000000477.07 rows=10 width=4) (actual time=2.271..3.263 rows=30 loops=1)
6	Group Key: employee_1.dno
7	Filter: (count(*) > 2)
8	-> Sort (cost=20000000439.19..20000000451.69 rows=5000 width=4) (actual time=2.222..2.623 rows=5000 loops=1)
9	Sort Key: employee_1.dno
10	Sort Method: quicksort Memory: 427kB
11	-> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.020..1.038 rows=5000 loops=1)
12	-> Index Only Scan using department_pkey on department (cost=0.14..1.35 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=30)
13	Index Cond: (dnumber = employee_1.dno)
14	Heap Fetches: 30
15	-> Index Scan using ehash on employee (cost=0.00..18.81 rows=42 width=4) (actual time=0.006..0.097 rows=42 loops=30)
16	Index Cond: (dno = department.dnumber)
17	Filter: (salary > 40000)
18	Rows Removed by Filter: 125
19	Planning Time: 0.376 ms
20	Execution Time: 6.853 ms

Estimated Cost:

.20000000685.46

1st Execution Time: 6.853 ms

2nd Execution Time: 7.683 ms

3rd Execution Time: 8.710 ms

Average Execution Time: 7.748 ms

Explanation: The reason behind this is that the hash-based index runs with time complexity O (1), which is much faster than O (n) without any indices and we can notice that in the execution time.

Best Scenario for Query 6:

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using Hash Index, since it has the least execution time.

Schema3

Modifications on data insertion:

- ***Populate sailors:***

Before:

```
public static void populateSailor(Connection conn) {
    for (int i = 1; i < 10000; i++) {
           if (insertSailor(i, "Sailor" + i,i,i, conn) == 0) {
               System.err.println("insertion of record " + i + " failed");
               break;
           } else
               System.out.println("insertion was successful");
       }
    }
```

After:

```
public static void populateSailor(Connection conn) {
    for (int i = 1; i <= 9000; i++) {
        if (insertSailor(i, "Sailor" + i,i,i, conn) == 0) {
            System.err.println("insertion of record " + i + " failed");
            break;
        } else
            System.out.println("insertion was successful");
    }
}
```

So, inserting 9000 sailors in the table

- ***Populate boat:***

Before:

```
public static void populateBoat(Connection conn) {
    for (int i = 1; i < 10000; i++) {
           if (insertBoat(i, "Boat" + i,"Red", conn) == 0) {
               System.err.println("insertion of record " + i + " failed");
               break;
           } else
               System.out.println("insertion was successful");
       }
    }
```

After:

```
public static void populateBoat(Connection conn) {
    for (int i = 1; i <= 3000; i++) {

        if(i>=1501) {
            if (insertBoat(i, "Boat" + i,"Green", conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
        else {
            if (insertBoat(i, "Boat" + i,"Red", conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }
}
```

So, inserting 3000 boat making half of them with red color and the other half with green color.

- ***Populate reserves:***

Before:

```
public static void populateReserves(Connection conn) {
    for (int i = 1; i < 10000; i++) {
        if (insertReserves(i, i,new Date(1,1,1999), conn) == 0) {
            System.err.println("insertion of record " + i + " failed");
            break;
        } else
            System.out.println("insertion was successful");
    }
}
```

After:

```
public static void populateReserves(Connection conn) {
    ArrayList<Pair> a= new ArrayList<>();
    for (int i = 1; i <= 15000; i++) {
        boolean duplicated=false;//not duplicated
        int bid=ThreadLocalRandom.current().nextInt(1,3001);
        int sid=ThreadLocalRandom.current().nextInt(1,9001);
        Pair TheNewPair=new Pair(sid, bid);
        for(int j=0;j<a.size();j++) {
            if((a.get(j)).compareTo(TheNewPair)==0){
                duplicated=true;//dup keys
                break;
            }
        }
        if(!duplicated) {
            a.add(TheNewPair);

            if (insertReserves(sid, bid,new Date(1,1,1999), conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }
}
```

```
class Pair{
    int a;
    int b;
    public Pair(int a,int b) {
        this.a=a;
        this.b=b;
    }
    public int  compareTo(Pair x) {
        if(this.a==x.a && this.b==x.b)
            return 0;
        else
            return -1;

    }
}
```

So, inserting 15000 reserves, creating an array list of pairs class I created , initializing duplicated flag to be false meaning that pair is unique so the primary key (sid and bid) is unique, intializing sid and bid variable to be random numbers with their maximum range as they are foreign keys, inserting in the pair sid and bid calling it TheNewPair Comparing it to all pairs in the arraylist if its duplicate flag is set to true. So, to let connection insert sid and bid the flag has to be false to prevent duplicate primary key.

Query7:

Output:

Sailor1496

Sailor3695

Sailor6330

---

without index:

set enable_indexscan=off;

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= off;

set enable_sort=on;

set enable_tidscan=off;

set enable_seqscan=on;

Explain Analyze Output:

"Nested Loop (cost=269.50..1657.02 rows=5 width=21) (actual time=15.317..51.659 rows=3 loops=1)"

" Join Filter: (s.sid = r.sid)"

" Rows Removed by Join Filter: 26997"

" -> Unique (cost=269.50..269.52 rows=5 width=4) (actual time=13.339..13.363 rows=3 loops=1)"

"   -> Sort (cost=269.50..269.51 rows=5 width=4) (actual time=13.334..13.342 rows=3 loops=1)"

"   Sort Key: r.sid"

"   Sort Method: quicksort Memory: 25kB"

"   -> Seq Scan on reserves r (cost=0.00..269.44 rows=5 width=4) (actual time=2.320..13.241 rows=3 loops=1)"

" Filter: (bid = 103)"

" Rows Removed by Filter: 14992"

" -> Seq Scan on sailors s (cost=0.00..165.00 rows=9000 width=25) (actual time=0.037..5.388 rows=9000 loops=3)"

"Planning time: 0.596 ms"

"Execution time: 51.828 ms"

1.Execution time1: 51.828 ms

2.Execution time2: 50.557 ms

3.Execution time3: 50.301 ms

Explanation:

Using seqsacn on both table sailors and reserves and using sort technique on table reserves then applying Unique on the results, then joining the results with sailors table using nested loop inner join with join filter s.sid=r.sid .

so, the cost was estimated to be: 1657.02 with an average execution time after running query 3 times equal (51.828+50.557+50.301)/3 = 50.895ms

with index:

Btree index:

*continuation to the previous stage

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= off;

set enable_sort=on;

set enable_tidscan=off;

create index r on reserves using btree(sid,bid);

create index s on sailors using btree(sid);

set enable_seqscan=off;

set enable_indexscan=on;

Explain Analyze Output:

"Nested Loop (cost=305.14..346.44 rows=5 width=21) (actual time=1.391..1.476 rows=3 loops=1)"

" -> Unique (cost=304.86..304.88 rows=5 width=4) (actual time=1.344..1.352 rows=3 loops=1)"

"   -> Sort (cost=304.86..304.87 rows=5 width=4) (actual time=1.341..1.344 rows=3 loops=1)"

"   Sort Key: r.sid"

"   Sort Method: quicksort Memory: 25kB"

"   -> Index Scan using r on reserves r (cost=0.29..304.80 rows=5 width=4) (actual time=0.273..1.312 rows=3 loops=1)"

"   Index Cond: (bid = 103)"

" -> Index Scan using s on sailors s (cost=0.29..8.30 rows=1 width=25) (actual time=0.029..0.033 rows=1 loops=3)"

"   Index Cond: (sid = r.sid)"

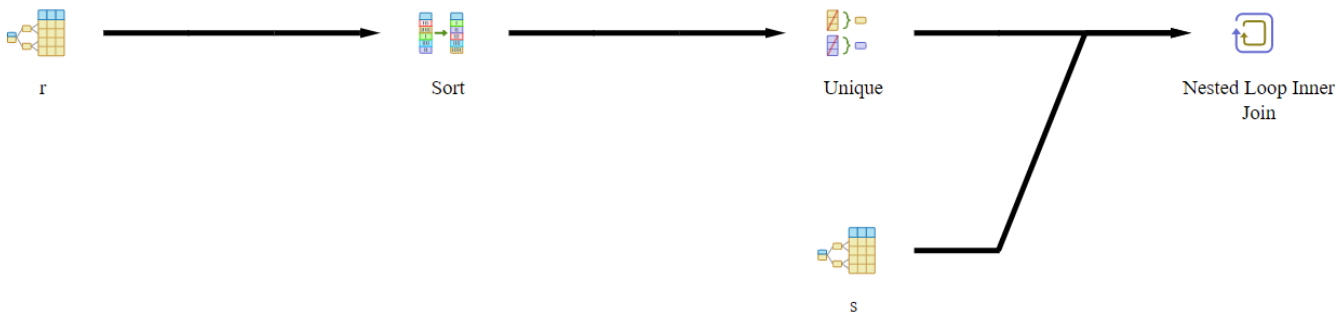
"Planning time: 0.665 ms"

"Execution time: 1.607 ms "

1.Execution time1: 1.607 ms

2.Execution time2: 2.202 ms

3.Execution time3: 1.808 ms



Explanation:

Using btree index on both reserves & sailors tables where r is an index created on the primary key of reserves (sid,bid) with condition(bid=103) then using sort technique on r (sort key is r.sid) then using unique on the results. s is an index created on the primary key of sailors (sid) with condition (sid=r.sid), then joining between results from Unique and s using nested inner loop join.

So, the cost was estimated to be: 346.44 with an average execution time after running query 3 times equal (1.607 + 2.202 + 1.808)/3 = 1.872 ms.so, we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 1.872 ms so this execution time is accepted

Hash index:

*continuation to the previous stage

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= off;

set enable_sort=on;

set enable_tidscan=off;

set enable_seqscan=off;

set enable_indexscan=on;

drop index r;

drop index s;

create index shash on sailors using hash(sid);

create index rshash on reserves using hash(sid);

create index rbhash on reserves using hash(bid);

alter table sailors alter column sid drop default;

Alter table sailors drop constraint sailors_pkey cascade;

Explain Analyze Output:

"Nested Loop (cost=24.15..64.31 rows=5 width=21) (actual time=0.152..0.195 rows=3 loops=1)"

" -> Unique (cost=24.15..24.17 rows=5 width=4) (actual time=0.121..0.127 rows=3 loops=1)"

"   -> Sort (cost=24.15..24.16 rows=5 width=4) (actual time=0.119..0.121 rows=3 loops=1)"

"   Sort Key: r.sid"

"   Sort Method: quicksort Memory: 25kB"

"   -> Index Scan using rbhash on reserves r (cost=0.00..24.09 rows=5 width=4) (actual time=0.059..0.084 rows=3 loops=1)"

"   Index Cond: (bid = 103)"

" -> Index Scan using shash on sailors s (cost=0.00..8.02 rows=1 width=25) (actual time=0.016..0.018 rows=1 loops=3)"

"   Index Cond: (sid = r.sid)"

"Planning time: 0.905 ms"

"Execution time: 0.336 ms"

1.Execution time1: 0.336 ms

2.Execution time2: 0.244 ms

3.Execution time3: 0.278 ms

Explanation:

Using hash-based index on both reserves & sailors tables where rshash is an index created on table reserves (sid) but query optimizer chooses rbhash index which is an index created on table reserves (bid) with condition(bid=103) then using Sort technique on rbhash (sort key is r.sid) then using Unique on the results. shash is an index created on the primary key of sailors (sid) with condition (sid=r.sid), then joining between results from Unique and shash using nested inner loop join.

So, the cost was estimated to be: 64.31 with an average execution time after running query 3 times equal (0.336 + 0.244 + 0.278)/3 = 0.286 ms. So, we notice that cost has decreased compared to without index case due to using hash-based index, as hash complexity is O(1), while without index is O(n) so the execution time also decreased to 0.286 ms so this execution time is accepted.

Bitmap index:

*continuation to the previous stage

set enable_indexonlyscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= off;

set enable_sort=on;

set enable_tidscan=off;

set enable_seqscan=off;

set enable_indexscan=on;

drop index rshash;

drop index shash;

drop index rbhash;

Set enable_bitmapscan=on;

create extension btree_gin;

create index sbitmap on sailors using gin(sid);

create index rbitmap on reserves using gin(sid,bid);

Explain Analyze Output:

"Nested Loop (cost=37.26..92.63 rows=5 width=21) (actual time=0.239..0.360 rows=3 loops=1)"

" -> Unique (cost=28.45..28.48 rows=5 width=4) (actual time=0.134..0.141 rows=3 loops=1)"

"   -> Sort (cost=28.45..28.47 rows=5 width=4) (actual time=0.132..0.134 rows=3 loops=1)"

"   Sort Key: r.sid"

"   Sort Method: quicksort Memory: 25kB"

"   -> Bitmap Heap Scan on reserves r (cost=12.04..28.40 rows=5 width=4) (actual time=0.086..0.104 rows=3 loops=1)"

"   Recheck Cond: (bid = 103)"

"   Heap Blocks: exact=3"

"   -> Bitmap Index Scan on rbitmap (cost=0.00..12.04 rows=5 width=0) (actual time=0.061..0.061 rows=3 loops=1)"

"   Index Cond: (bid = 103)"

" -> Bitmap Heap Scan on sailors s (cost=8.81..12.82 rows=1 width=25) (actual time=0.041..0.042 rows=1 loops=3)"

"   Recheck Cond: (sid = r.sid)"

"   Heap Blocks: exact=3"

"   -> Bitmap Index Scan on sbitmap (cost=0.00..8.81 rows=1 width=0) (actual time=0.029..0.029 rows=1 loops=3)"

"   Index Cond: (sid = r.sid)"

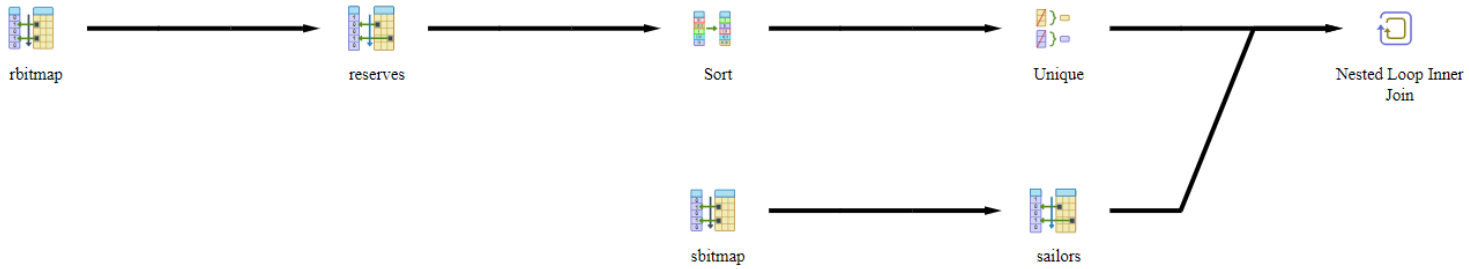
"Planning time: 0.937 ms"

"Execution time: 0.636 ms"

1.Execution time1: 0.636 ms

2.Execution time2: 0.546 ms

3 Execution time3: 0.352 ms



Explanation:

Using bitmap index on both reserves & sailors tables where rbitmap index is an index created primary key of table reserves (sid,bid) with condition (bid=103) then bitmap heap scan is done with recheck condition (bid = 103) then using Sort technique (sort key is r.sid) then using Unique on the results, sbitmap is an index created on the primary key of sailors (sid) with condition (sid=r.sid) then bitmap heap scan is done with recheck condition(sid=r.sid), then joining between the results using nested inner loop join.

So, the cost was estimated to be: 92.63 with an average execution time after running query 3 times equal (0.636 + 0.546 + 0.278)/3= 0.486ms. So, we notice that cost has decreased compared to without index case due to using bitmap index, as it makes the operation faster so the execution time also decreased to 0.486 ms so this execution time is accepted

The best scenario:

	Without index	btree	hash	bitmap
Cost	1657.02	346.44	64.31	92.63
Exec.	50.895	1.872	0.286	0.486

So, hash-based index is the best index for this query.

Query8:

Output:

Sample from the output:

"Sailor1489 "Sailor4790"Sailor273"Sailor2574"Sailor5761"Sailor5843

"Sailor7662"Sailor5259"Sailor2466"Sailor2196"

----

without index:

*continuation to the previous stage/s

set enable_indexscan=off;

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

set enable_tidscan=on;

set enable_nestloop=on;

set enable_sort=on;

set enable_seqscan=on;

Explain Analyze Output:

"Nested Loop (cost=6348.36..27161.61 rows=75 width=21) (actual time=23187.612..63777.254 rows=5114 loops=1)"

" Join Filter: (s.sid = r.sid)"

" Rows Removed by Join Filter: 46020886"

" -> HashAggregate (cost=6348.36..6349.11 rows=75 width=4) (actual time=23174.715..23196.845 rows=5114 loops=1)"

"   Group Key: r.sid"

" -> Nested Loop (cost=56.54..6348.18 rows=75 width=4) (actual time=5.867..23133.845 rows=7485 loops=1)"

" Join Filter: (r.bid = b.bid)"

" Rows Removed by Join Filter: 22488015"

" -> HashAggregate (cost=56.54..56.69 rows=15 width=4) (actual time=5.649..11.710 rows=1500 loops=1)"

" Group Key: b.bid"

" -> Seq Scan on boat b (cost=0.00..56.50 rows=15 width=4) (actual time=0.101..2.860 rows=1500 loops=1)"

" Filter: ((color)::text = 'Red'::text)"

" Rows Removed by Filter: 1500"

" -> Seq Scan on reserves r (cost=0.00..231.97 rows=14997 width=8) (actual time=0.021..5.832 rows=14997 loops=1500)"

" -> Seq Scan on sailors s (cost=0.00..165.00 rows=9000 width=25) (actual time=0.013..3.149 rows=9000 loops=5114)"

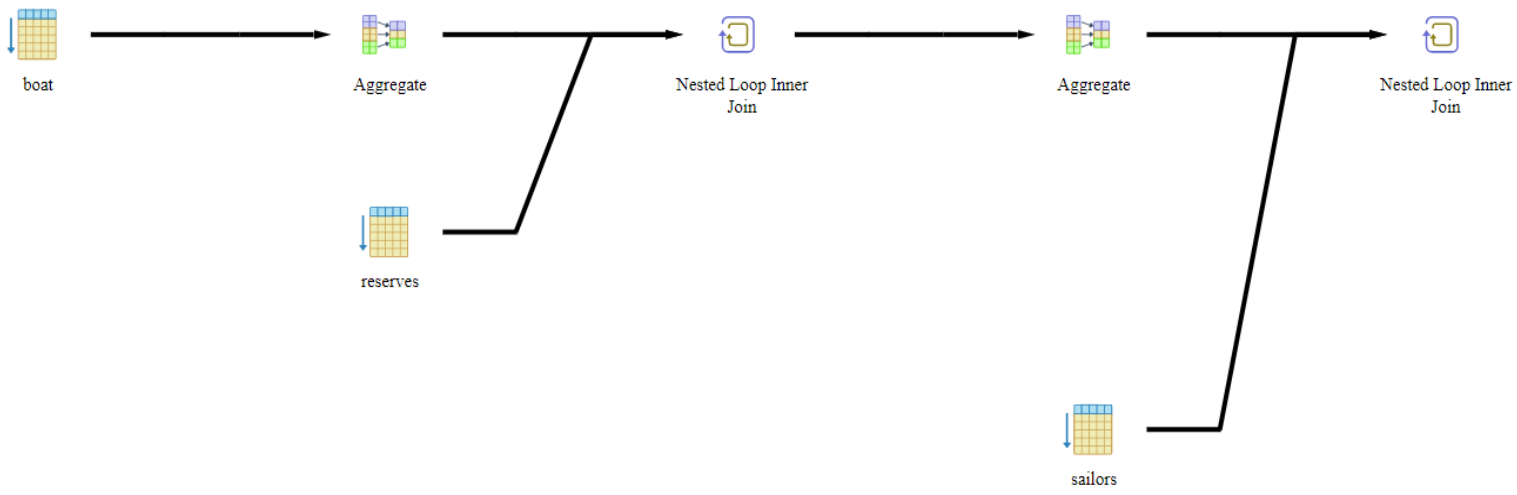
"Planning time: 4.234 ms"

"Execution time: 63782.609 ms"

1.Execution time1: 63782.609 ms

2.Execution time2: 69193.483 ms

3.Execution time3: 69109.816 ms



Explanation:

Using seqsacn on tables boat, sailors & reserves.Applying Hash Aggregate on table boat (Group Key: b.bid) then joining between results of table reserves and of hash Agg. using nested loop inner join with join filter (r.bid = b.bid), then applying hash Agg. on the results with (Group Key: r.sid), then joining the results with sailors table using nested loop inner join with join filter (s.sid = r.sid).

So, the cost was estimated to be: 27161.61 with an average execution time after running query 3 times equal (63782.609 + 69193.483 + 69109.816)/3 = 67361.969 ms.

with index:

Btree index:

*continuation to the previous stage/s

```
set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

set enable_tidscan=on;

set enable_nestloop=on;

set enable_sort=on;

set enable_indexscan=on;

set enable_seqscan=off;

alter table boat alter column bid drop default;

Alter table boat drop constraint boat_pkey cascade;

create index r on reserves using btree(sid,bid);

create index s on sailors using btree (sid);

create index b on boat using btree(bid);

create index bc on boat using btree(color);
```

Explain Analyze Output:

"Nested Loop (cost=2122.14..2150.22 rows=75 width=21) (actual time=3124.419..3194.066 rows=5114 loops=1)"

" -> HashAggregate (cost=2121.85..2122.60 rows=75 width=4) (actual time=3124.367..3133.882 rows=5114 loops=1)"

"   Group Key: r.sid"

"   -> Nested Loop (cost=48.86..2121.67 rows=75 width=4) (actual time=4.679..3088.164 rows=7485 loops=1)"

"   -> HashAggregate (cost=48.58..48.73 rows=15 width=4) (actual time=4.464..10.020 rows=1500 loops=1)"

"   Group Key: b.bid"

"   -> Index Scan using bc on boat b (cost=0.28..48.54 rows=15 width=4) (actual time=0.231..1.793 rows=1500 loops=1)"

"   Index Cond: ((color)::text = 'Red'::text)"

"   -> Index Scan using r on reserves r (cost=0.29..138.15 rows=5 width=8) (actual time=0.476..2.034 rows=5 loops=1500)"

"   Index Cond: (bid = b.bid)"

" -> Index Scan using s on sailors s (cost=0.29..0.36 rows=1 width=25) (actual time=0.008..0.010 rows=1 loops=5114)"

"   Index Cond: (sid = r.sid)"

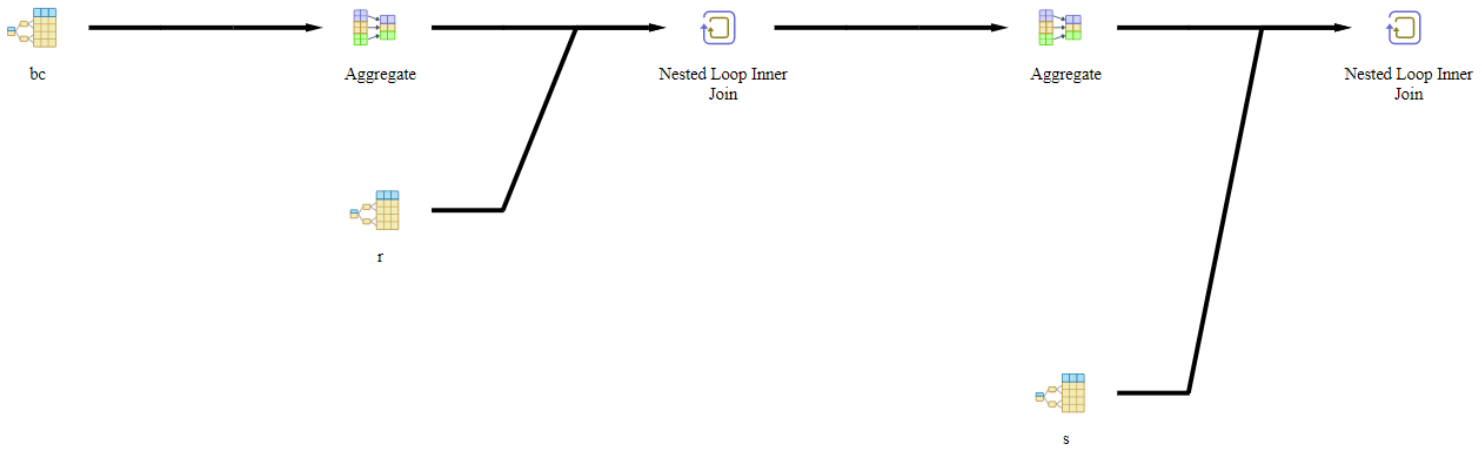
"Planning time: 2.518 ms"

"Execution time: 3199.089 ms"

1.Execution time1: 3199.089 ms"

2.Execution time2: 2969.681 ms

3.Execution time3: 3523.717 ms



Explanation:

Using btree index on tables boat, sailors & reserves where b is an index created on primary key of table boat but the query optimizer chooses bc index which created on column color in boat table with Index Cond: ((color)::text = 'Red'::text), then applying Hash Aggregate on result of index scan (Group Key: b.bid). r is an index created on the primary key of table reserves (sid,bid) with Index Cond: (bid = b.bid), then joining between results of r and of hash Agg. using nested loop inner join, then applying hash Agg. on the results. s is an index created on table sailors with Index Cond: (sid = r.sid),joining the results from hash Agg. with results from s using nested loop inner join.

So, the cost was estimated to be: 2150.22 with an average execution time after running query 3 times equal (3199.089 + 2969.681 + 3523.717)/3 = 3230.829 ms. we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 3230.829 ms so this execution time is accepted.

Hash index:

*continuation to the previous stage/s

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

```
set enable_tidscan=on;
set enable_nestloop=on;
set enable_sort=on;
set enable_indexscan=on;
set enable_seqscan=off;
drop index b;
drop index r;
drop index s;
create index bhash on boat using hash(bid);
create index rshash on reserves using hash(sid);
create index rbhash on reserves using hash(bid);
create index shash on sailors using hash(sid);
create index bchash on boat using hash(color);
```

Explain Analyze Output:

```
"Nested Loop  (cost=314.70..321.97 rows=75 width=21) (actual time=60.355..118.219 rows=5114 loops=1)"
"  ->  HashAggregate  (cost=314.70..315.45 rows=75 width=4) (actual time=60.336..65.321 rows=5114 loops=1)"
"        Group Key: r.sid"
"        ->  Nested Loop  (cost=48.30..314.51 rows=75 width=4) (actual time=6.077..47.872 rows=7485 loops=1)"
"              ->  HashAggregate  (cost=48.30..48.45 rows=15 width=4) (actual time=6.043..8.038 rows=1500 loops=1)"
"                    Group Key: b.bid"
"                    ->  Index Scan using bchash on boat b  (cost=0.00..48.26 rows=15 width=4) (actual time=0.045..3.324 rows=1500 loops=1)"
"                          Index Cond: ((color)::text = 'Red'::text)"
"              ->  Index Scan using rbhash on reserves r  (cost=0.00..17.69 rows=5 width=8) (actual time=0.006..0.022 rows=5 loops=1500)"
"                    Index Cond: (bid = b.bid)"
"  ->  Index Scan using shash on sailors s  (cost=0.00..0.08 rows=1 width=25) (actual time=0.007..0.008 rows=1 loops=5114)"
```

" Index Cond: (sid = r.sid)"

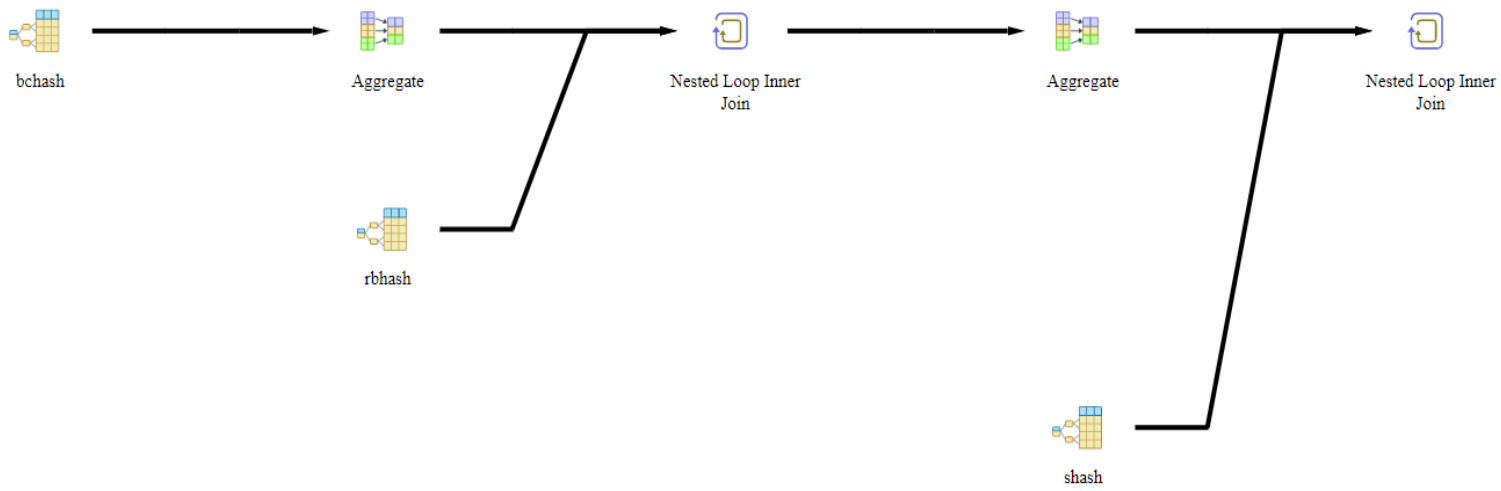
"Planning time: 0.835 ms"

"Execution time: 120.492 ms"

1.Execution time1: 120.492 ms

2.Execution time2: 185.671 ms

3.Execution time: 142.391 ms



Explanation:

Using hash-based index on tables boat, sailors & reserves where bhash is an index created on the primary key of table boat but the query optimizer chooses bchash which is an index created on column color from table boat with Index Cond: ((color)::text = 'Red'::text)", then applying Hash Aggregate on the results from bchash(Group Key: b.bid). rshash is index created on table reserves(sid) but the optimizer chooses rbhash which is an index created on table reserves(bid) with Index Cond: (bid = b.bid). Then joining between results of table rbhash and of hash Agg. using nested loop inner join, then applying hash Agg. on the results with (Group Key: r.sid). shahsh is an index created on the primary key of table sailors(sid) with Index Cond: (sid = r.sid). Then joining the results of shash and of hash Agg. using nested loop inner join.

So, the cost was estimated to be: 321.97 with an average execution time after running query 3 times equal (120.492 + 185.671 + 142.391)/3 = 149.518 ms. So, we notice that cost has decreased compared to without index case due to using Hash-based bindex, as hash complexity is O(1) while without index is O(n) so the execution time also decreased to 149.518 ms so this execution time is accepted.

Bitmap index:

*continuation to the previous stage/s

```
set enable_indexonlyscan=off;
set enable_hashjoin=off;
set enable_mergejoin=off;
set enable_material=off;
set enable_hashagg= on;
set enable_tidscan=on;
set enable_nestloop=on;
set enable_sort=on;
set enable_indexscan=on;
set enable_seqscan=off;
set enable_bitmapscan=on;
drop index shash;
drop index bhash;
drop index bchash;
drop index rshash;
drop index rbhash;
create extension btree_gin;
create index sbitmap on sailors using gin(sid);
create index rbitmap on reserves using gin(sid,bid);
create index bbitmap on boat using gin(bid);
create index bcitmap on boat using gin(color);
```

Explain Analyze Output:

```
"Nested Loop  (cost=338.13..643.68 rows=75 width=21) (actual time=134.217..571.313 rows=5114 loops=1)"
"  ->  HashAggregate  (cost=338.09..338.84 rows=75 width=4) (actual time=134.175..146.042 rows=5114 loops=1)"
```

" Group Key: r.sid"

" -> Nested Loop (cost=34.47..337.90 rows=75 width=4) (actual time=5.562..119.831 rows=7485 loops=1)"

" -> HashAggregate (cost=27.23..27.38 rows=15 width=4) (actual time=5.418..7.277 rows=1500 loops=1)"

" Group Key: b.bid"

" -> Bitmap Heap Scan on boat b (cost=8.12..27.19 rows=15 width=4) (actual time=1.093..2.082 rows=1500 loops=1)"

" Recheck Cond: ((color)::text = 'Red'::text)"

" Heap Blocks: exact=9"

" -> Bitmap Index Scan on bcitmap (cost=0.00..8.11 rows=15 width=0) (actual time=0.960..0.960 rows=1500 loops=1)"

" Index Cond: ((color)::text = 'Red'::text)"

" -> Bitmap Heap Scan on reserves r (cost=7.24..20.65 rows=5 width=8) (actual time=0.028..0.041 rows=5 loops=1500)"

" Recheck Cond: (bid = b.bid)"

" Heap Blocks: exact=7270"

" -> Bitmap Index Scan on rbitmap (cost=0.00..7.24 rows=5 width=0) (actual time=0.018..0.018 rows=5 loops=1500)"

" Index Cond: (bid = b.bid)"

" -> Bitmap Heap Scan on sailors s (cost=0.04..4.05 rows=1 width=25) (actual time=0.036..0.036 rows=1 loops=5114)"

" Recheck Cond: (sid = r.sid)"

" Heap Blocks: exact=5114"

" -> Bitmap Index Scan on sbitmap (cost=0.00..0.04 rows=1 width=0) (actual time=0.019..0.019 rows=1 loops=5114)"

" Index Cond: (sid = r.sid)"

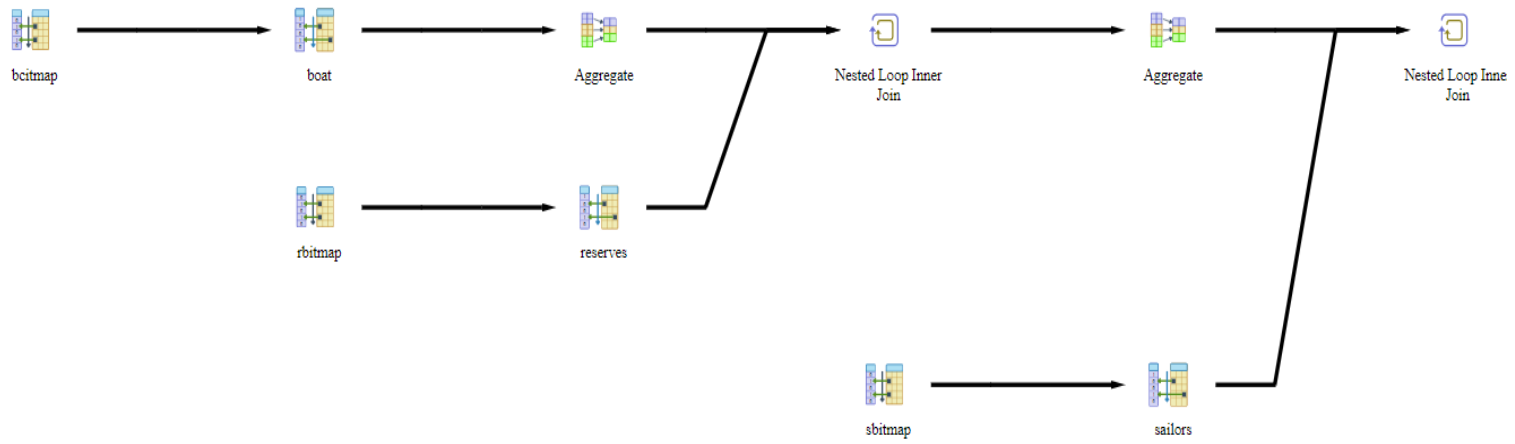
"Planning time: 1.381 ms"

"Execution time: 574.542 ms"

1.Execution time1: 574.542 ms"

2.Execution time2: 721.748 ms

3.Execution time3: 445.890 ms



Explanation:

Using bitmap index on both reserves & sailors tables where rbitmap index is an index created primary key of table boat but the query optimizer chooses bcitmap which is an index created on the column color of table boat with Index Cond: ((color)::text = 'Red'::text) then bitmap heap scan is done on reserves r with Recheck Cond: (bid = b.bid) then appyling Hash aggreate on the results with Group Key: b.bid. rbitmap is an index created on the primary key of table reserves(sid,bid) with Index Cond: (bid = b.bid) , then applying bitmap heap scan on sailors s with Recheck Cond: (bid = b.bid), joining the results from hash Agg. And bitmap heap scan using nested loop inner join, then applyin hash agg. on the results with Group Key: r.sid. sbitmap is an index created on the primary key of sailors (sid) with Index Cond: (sid = r.sid) then bitmap heap scan is doneon table sailors s with Recheck Cond: (sid = r.sid),then joining the results from bitmap heap scan and hash Ag. using nested inner loop join.

So, the cost was estimated to be: 643.68 with an average execution time after running query 3 times equal (574.542 + 721.748 + 445.890)/3= 580.726 ms. So, we notice that cost has decreased to compared to without index case due to using bitmap index, as it makes the operation faster so the execution time also decreased to 580.726 ms so this execution time is accepted.

The best scenario:

	Without index	btree	hash	bitmap
Cost	27161.61	2150.22	321.97	643.68
Exec.	67361.969	3230.829	149.518	580.726

So, hash-based index is the best index for this query.

Query9:

Output:

Sample from the output:

"Sailor5496"Sailor2788"Sailor232"Sailor2825
"Sailor8526"Sailor2887"Sailor197"Sailor8795 "Sailor7515”
"Sailor7769"Sailor7261"Sailor4867"Sailor4579 "

----

without index:

*continuation to the previous stage/s

set enable_indexscan=off;

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

set enable_tidscan=off;

set enable_nestloop=on;

set enable_sort=on;

set enable_seqscan=on;

Explain Analyze Output:

"Nested Loop (cost=27160.67..67569.55 rows=1 width=21) (actual time=81520.406..217222.988 rows=4304 loops=1)"

" Join Filter: (r.sid = s.sid)"

" Rows Removed by Join Filter: 38731696"

" -> Nested Loop (cost=27160.67..67292.05 rows=1 width=12) (actual time=81516.864..182813.826 rows=4304 loops=1)"

"   Join Filter: (r.bid = b.bid)"

" Rows Removed by Join Filter: 17719696"

" -> Nested Loop (cost=27160.67..58618.86 rows=153 width=16) (actual time=81509.624..148443.366 rows=11816 loops=1)"

" Join Filter: (r.sid = s2.sid)"

" Rows Removed by Join Filter: 76997779"

" -> HashAggregate (cost=27160.67..27161.42 rows=75 width=8) (actual time=81501.302..81534.230 rows=5135 loops=1)"

" Group Key: s2.sid"

" -> Nested Loop (cost=0.00..27160.49 rows=75 width=8) (actual time=7.058..81440.149 rows=7512 loops=1)"

" Join Filter: (r2.sid = s2.sid)"

" Rows Removed by Join Filter: 67600488"

" -> Nested Loop (cost=0.00..6347.99 rows=75 width=4) (actual time=0.757..20874.778 rows=7512 loops=1)"

" Join Filter: (b2.bid = r2.bid)"

" Rows Removed by Join Filter: 22487988"

" -> Seq Scan on boat b2 (cost=0.00..56.50 rows=15 width=4) (actual time=0.491..6.399 rows=1500 loops=1)"

" Filter: ((color)::text = 'Green'::text)"

" Rows Removed by Filter: 1500"

" -> Seq Scan on reserves r2 (cost=0.00..231.97 rows=14997 width=8) (actual time=0.016..5.332 rows=14997 loops=1500)"

" -> Seq Scan on sailors s2 (cost=0.00..165.00 rows=9000 width=4) (actual time=0.013..3.251 rows=9000 loops=7512)"

" -> Seq Scan on reserves r (cost=0.00..231.97 rows=14997 width=8) (actual time=0.017..5.130 rows=14997 loops=5135)"

" -> Seq Scan on boat b (cost=0.00..56.50 rows=15 width=4) (actual time=0.019..2.199 rows=1500 loops=11816)"

" Filter: ((color)::text = 'Red'::text)"

" Rows Removed by Filter: 1500"

" -> Seq Scan on sailors s (cost=0.00..165.00 rows=9000 width=25) (actual time=0.012..3.266 rows=9000 loops=4304)"

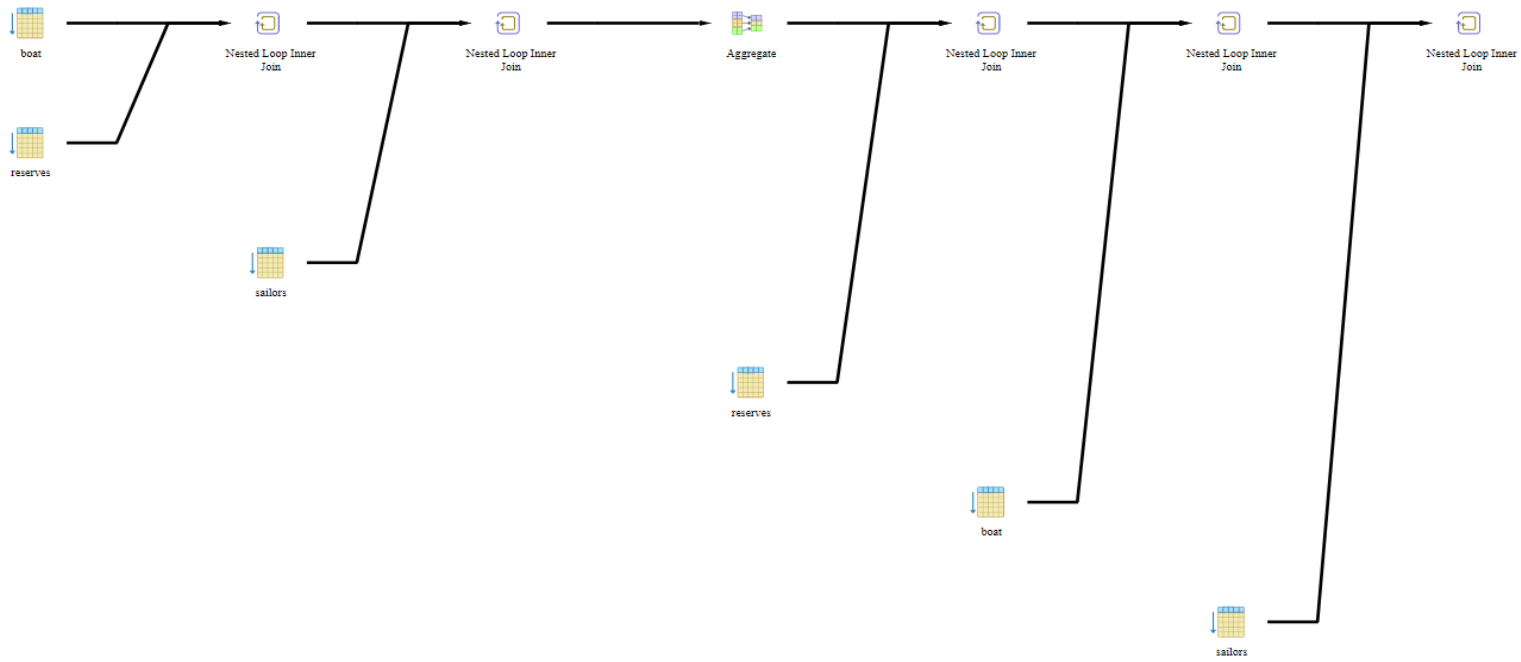
"Planning time: 0.767 ms"

"Execution time: 217229.237 ms"

1.Execution time1: 217229.237 ms

2.Execution time2: 223701.459 ms

3.Execution time3: 225569.623 ms



Explanation:

Using seqsacn on tables boat, sailors & reserves. Joining between seqscan of table boat b2 with (Filter: ((color)::text = 'Green'::text)") and reserves r2 using nestloop inner join with Join Filter: (b2.bid = r2.bid), then joining between results from nestedloop and results from seqscan on table sailors s2 using nestedloop inner join with Join Filter: (r2.sid = s2.sid), then applying hash Aggregate on the results with Group Key: s2.sid, then joining the results with the seqscan on table reserves r using nestedloop inner join with Join Filter: (r.sid = s2.sid),then joiningthe results with the results of seqscan on table boat b with Filter: ((color)::text = 'Red'::text)" using nestedloop inner join with Join Filter: (r.bid = b.bid), then joining results with results of seqscn on table sailors s using nestedloop inner join with Join Filter: (r.sid = s.sid).

So, the cost was estimated to be: 67569.55 with an average execution time after running query 3 times equal (217229.237 + 223701.459 + 225569.623)/3 = 222166.773 ms.

with index:

Btree index:

*continuation to the previous stage/s

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

set enable_tidscan=off;

set enable_nestloop=on;

set enable_sort=on;

set enable_indexscan=on;

set enable_seqscan=off;

create index r on reserves using btree(sid,bid);

create index s on sailors using btree (sid);

create index b on boat using btree(bid);

create index bc on boat using btree(color);

Explain Analyze Output:

"Nested Loop (cost=1.70..2226.02 rows=1 width=21) (actual time=0.916..4129.699 rows=4304 loops=1)"

" -> Nested Loop (cost=0.57..2121.48 rows=75 width=4) (actual time=0.463..3323.328 rows=7485 loops=1)"

"   -> Index Scan using bc on boat b (cost=0.28..48.54 rows=15 width=4) (actual time=0.102..9.001 rows=1500 loops=1)"

"   Index Cond: ((color)::text = 'Red'::text)"

"   -> Index Scan using r on reserves r (cost=0.29..138.15 rows=5 width=8) (actual time=0.463..2.198 rows=5 loops=1500)"

"   Index Cond: (bid = b.bid)"

" -> Nested Loop Semi Join (cost=1.14..1.38 rows=1 width=33) (actual time=0.101..0.104 rows=1 loops=7485)"

"   -> Index Scan using s on sailors s (cost=0.29..0.33 rows=1 width=25) (actual time=0.017..0.020 rows=1 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Nested Loop (cost=0.85..1.04 rows=1 width=8) (actual time=0.079..0.079 rows=1 loops=7485)"

"   -> Nested Loop (cost=0.57..0.73 rows=1 width=12) (actual time=0.027..0.038 rows=2 loops=7485)"

"   Join Filter: (s2.sid = r2.sid)"

"   -> Index Scan using s on sailors s2 (cost=0.29..0.35 rows=1 width=4) (actual time=0.009..0.010 rows=1 loops=7485)"

"   Index Cond: (sid = s.sid)"

"   -> Index Scan using r on reserves r2 (cost=0.29..0.35 rows=2 width=8) (actual time=0.012..0.020 rows=2 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Index Scan using b on boat b2 (cost=0.28..0.31 rows=1 width=4) (actual time=0.013..0.013 rows=0 loops=17863)"

"   Index Cond: (bid = r2.bid)"

"   Filter: ((color)::text = 'Green'::text)"

"   Rows Removed by Filter: 1"

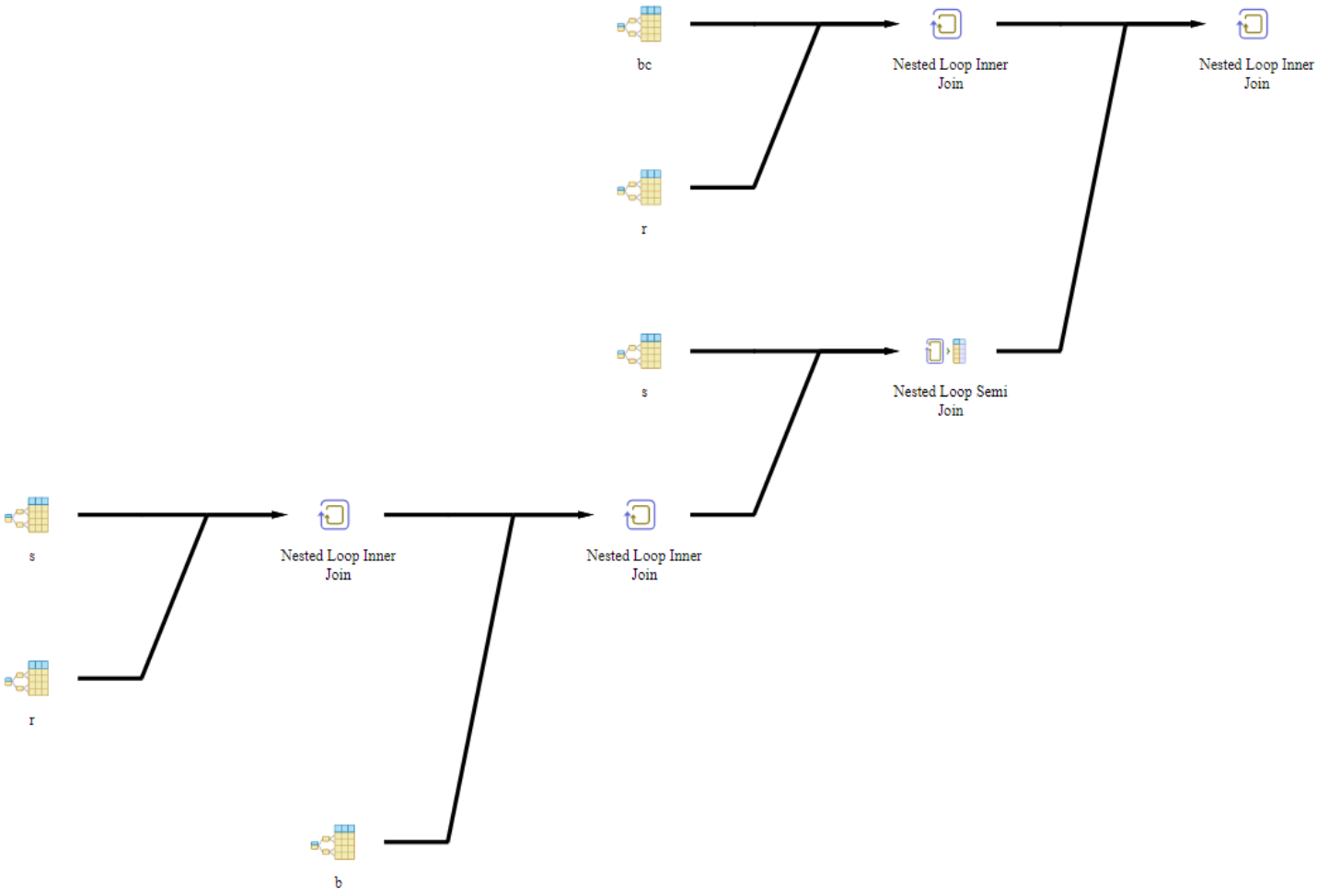
"Planning time: 2.090 ms"

"Execution time: 4132.226 ms"

1.Execution time1: 4132.226 ms"

2.Execution time2: 3485.928 ms

3.Execution time3: 3461.267 ms



Explanation:

Using btree index on tables boat, reserves & sailors, where s is an index created on the primary key column of table sailors(sid) , r is an index created on the column primary key of the table reserves (sid.bid), b is an index created on the column primary key of table boat (bid)& bc is an index created on table boat using color column. So, joining between results of index r on reserves 2 with Index Cond: (sid = r.sid) and the results of index s on sailors s2 with Index Cond: (sid = s.sid) using neestedloop inner join Join Filter: (s2.sid = r2.sid) ,then joining the results with the results of index b on boat 2 with Index Cond: (bid = r2.bid) & Filter: ((color)::text = 'Green'::text) using neested loop inner join, then joining the results with results ,(from index s on sailors s with Index Cond: (sid = r.sid) ,then joining the results with the results from joining the results of index bc on boat b with Index Cond: ((color)::text = 'Red'::text) and results from index r on reserves r with Index Cond: (bid = b.bid) using neested loop inner join) ,using neestedloop inner join.

So, the cost was estimated to be: 2226.02 with an average execution time after running query 3 times equal (4132.226 + 3485.928 + 3461.267)/3 = 3693.140 ms.so, we notice that cost has

decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 3693.140 ms so this execution time is accepted

Hash index:

*continuation to the previous stage/s

set enable_indexscan=on;

set enable_indexonlyscan=off;

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

set enable_material=off;

set enable_hashagg= on;

set enable_tidscan=off;

set enable_nestloop=on;

set enable_sort=on;

set enable_seqscan=off;

drop index r;

drop index s;

drop index b;

drop index bc;

create index bhash on boat using hash(bid);

create index bchash on boat using hash(color);

create index shahsh on sailors using hash(sid);

create index rshahsh on reserves using hash(sid);

create index rbhahsh on reserves using hash(bid);

Explain Analyze Output:

"Nested Loop (cost=0.00..334.71 rows=1 width=21) (actual time=0.314..572.525 rows=4304 loops=1)"

" -> Nested Loop (cost=0.00..314.32 rows=75 width=4) (actual time=0.074..68.258 rows=7485 loops=1)"

"   -> Index Scan using bchash on boat b (cost=0.00..48.26 rows=15 width=4) (actual time=0.044..5.457 rows=1500 loops=1)"

"   Index Cond: ((color)::text = 'Red'::text)"

"   -> Index Scan using rbhahsh on reserves r (cost=0.00..17.69 rows=5 width=8) (actual time=0.009..0.037 rows=5 loops=1500)"

"   Index Cond: (bid = b.bid)"

" -> Nested Loop Semi Join (cost=0.00..0.26 rows=1 width=33) (actual time=0.063..0.065 rows=1 loops=7485)"

"   -> Index Scan using shahsh on sailors s (cost=0.00..0.05 rows=1 width=25) (actual time=0.009..0.012 rows=1 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Nested Loop (cost=0.00..0.21 rows=1 width=8) (actual time=0.049..0.049 rows=1 loops=7485)"

"   -> Nested Loop (cost=0.00..0.17 rows=1 width=12) (actual time=0.020..0.028 rows=2 loops=7485)"

"   Join Filter: (s2.sid = r2.sid)"

"   -> Index Scan using shahsh on sailors s2 (cost=0.00..0.07 rows=1 width=4) (actual time=0.006..0.007 rows=1 loops=7485)"

"   Index Cond: (sid = s.sid)"

"   -> Index Scan using rshahsh on reserves r2 (cost=0.00..0.07 rows=2 width=8) (actual time=0.009..0.013 rows=2 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Index Scan using bhash on boat b2 (cost=0.00..0.03 rows=1 width=4) (actual time=0.009..0.009 rows=0 loops=13439)"

"   Index Cond: (bid = r2.bid)"

"   Filter: ((color)::text = 'Green'::text)"

"   Rows Removed by Filter: 1"

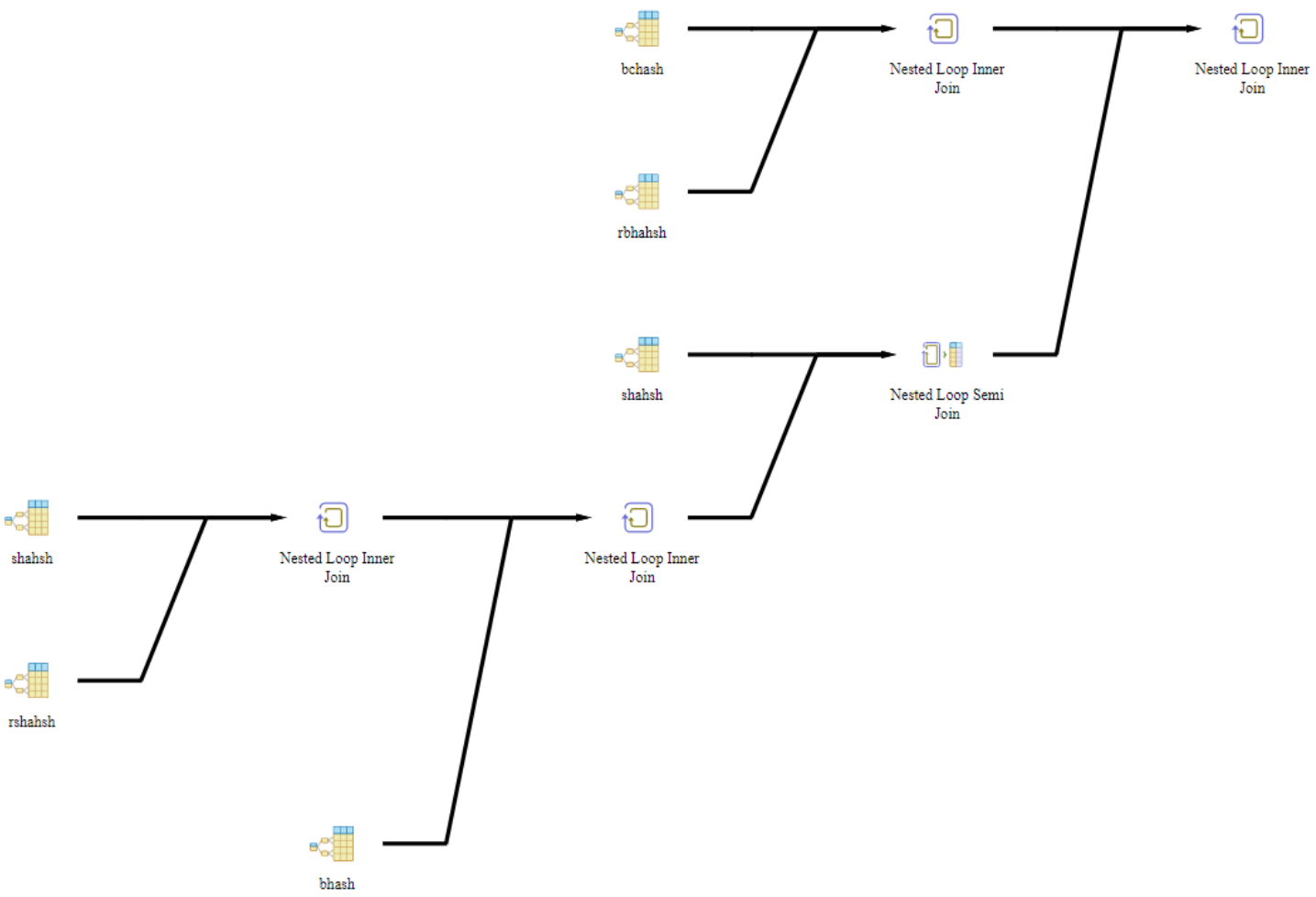
"Planning time: 59.806 ms"

"Execution time: 574.619 ms"

1.Execution time1: 574.619 ms

2.Execution time2: 555.017 ms

3.Execution time3: 573.221 ms



Explanation:

Using hash-based index on tables boat, reserves & sailors, where shahsh is an index created on the primary key column of table sailors(sid) , rshahsh is an index created on table reserves (sid), rbhahsh is an index created on table reserves (bid), bhash is an index created on the primary key column of table boat (bid)& bchash is an index created on table boat using color column. Joining between results of index shahsh on sailors s2, with Index Cond:(sid=s.sid),With the results of index rshahsh on reserves r2 with Index Cond: (sid = r.sid) using neestedloop inner join with Join Filter: (s2.sid = r2.sid), then joining the results with the results of index bhash on boat b2 with Index Cond: (bid = r2.bid) 1& Filter:((color)::text ='Green'::text)) using neested loop inner join then joining the results with the results of index shahsh on sailors s,with Index Cond: (sid = r.sid),using neestedloop semi join then joinig the results with(the result of joining of index bchash on boat b ,with Index Cond: ((color)::text = 'Red'::text), with results of index rbhahsh on reserves r , with Index Cond: (bid = b.bid),using neestedloop inner join),using neestedloop inner join.

So, the cost was estimated to be: 334.71 with an average execution time after running query 3 times equal (574.619 + 555.017 + 573.221)/3 = 567.619 ms. So, we notice that cost has decreased compared to without index case due to using Hash-based bindex, as hash complexity is O(1) while without index is O(n) so the execution time also decreased to 606.952 ms so this execution time is accepted.

Bitmap index:

*continuation to the previous stage/s

```
set enable_indexscan=on;
set enable_indexonlyscan=off;
set enable_hashjoin=off;
set enable_mergejoin=off;
set enable_material=off;
set enable_hashagg= on;
set enable_tidscan=off;
set enable_nestloop=on;
set enable_sort=on;
set enable_seqscan=off;
set enable_bitmapscan=on;
drop index bchash;
 drop index shahsh;
 drop index bhash;
 drop index rshahsh;
 drop index rbhahsh;
 create index bbitmap on boat using gin(bid);
 create index bcbitmap on boat using gin(color);
 create index sbitmap on sailors using gin(sid);
 create index rbitmap on reserves using gin(sid,bid);
```

Explain Analyze Output:

"Nested Loop (cost=15.47..1555.26 rows=1 width=21) (actual time=1.380..2282.308 rows=4304 loops=1)"

" -> Nested Loop (cost=15.36..337.71 rows=75 width=4) (actual time=1.096..135.990 rows=7485 loops=1)"

"   -> Bitmap Heap Scan on boat b (cost=8.12..27.19 rows=15 width=4) (actual time=0.991..1.933 rows=1500 loops=1)"

"   Recheck Cond: ((color)::text = 'Red'::text)"

"   Heap Blocks: exact=9"

"   -> Bitmap Index Scan on bcbitmap (cost=0.00..8.11 rows=15 width=0) (actual time=0.935..0.936 rows=1500 loops=1)"

"   Index Cond: ((color)::text = 'Red'::text)"

"   -> Bitmap Heap Scan on reserves r (cost=7.24..20.65 rows=5 width=8) (actual time=0.034..0.052 rows=5 loops=1500)"

"   Recheck Cond: (bid = b.bid)"

"   Heap Blocks: exact=7270"

"   -> Bitmap Index Scan on rbitmap (cost=0.00..7.24 rows=5 width=0) (actual time=0.023..0.023 rows=5 loops=1500)"

"   Index Cond: (bid = b.bid)"

" -> Nested Loop Semi Join (cost=0.11..16.22 rows=1 width=33) (actual time=0.282..0.283 rows=1 loops=7485)"

"   -> Bitmap Heap Scan on sailors s (cost=0.02..4.04 rows=1 width=25) (actual time=0.023..0.024 rows=1 loops=7485)"

"   Recheck Cond: (sid = r.sid)"

"   Heap Blocks: exact=7485"

"   -> Bitmap Index Scan on sbitmap (cost=0.00..0.02 rows=1 width=0) (actual time=0.017..0.017 rows=1 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Nested Loop (cost=0.09..12.18 rows=1 width=8) (actual time=0.225..0.225 rows=1 loops=7485)"

"   -> Nested Loop (cost=0.08..8.14 rows=1 width=12) (actual time=0.117..0.123 rows=2 loops=7485)"

"   Join Filter: (s2.sid = r2.sid)"

"   -> Bitmap Heap Scan on sailors s2 (cost=0.04..4.05 rows=1 width=4) (actual time=0.020..0.020 rows=1 loops=7485)"

"   Recheck Cond: (sid = s.sid)"

"   Heap Blocks: exact=7485"

"   -> Bitmap Index Scan on sbitmap (cost=0.00..0.04 rows=1 width=0) (actual time=0.014..0.014 rows=1 loops=7485)"

"   Index Cond: (sid = s.sid)"

"   -> Bitmap Heap Scan on reserves r2 (cost=0.04..4.07 rows=2 width=8) (actual time=0.036..0.039 rows=2 loops=7485)"

"   Recheck Cond: (sid = r.sid)"

"   Heap Blocks: exact=13339"

"   -> Bitmap Index Scan on rbitmap (cost=0.00..0.04 rows=2 width=0) (actual time=0.021..0.021 rows=3 loops=7485)"

"   Index Cond: (sid = r.sid)"

"   -> Bitmap Heap Scan on boat b2 (cost=0.01..4.03 rows=1 width=4) (actual time=0.022..0.022 rows=0 loops=13468)"

"   Recheck Cond: (bid = r2.bid)"

"   Filter: ((color)::text = 'Green'::text)"

"   Rows Removed by Filter: 1"

"   Heap Blocks: exact=13468"

"   -> Bitmap Index Scan on bbitmap (cost=0.00..0.01 rows=1 width=0) (actual time=0.014..0.014 rows=1 loops=13468)"

"   Index Cond: (bid = r2.bid)"

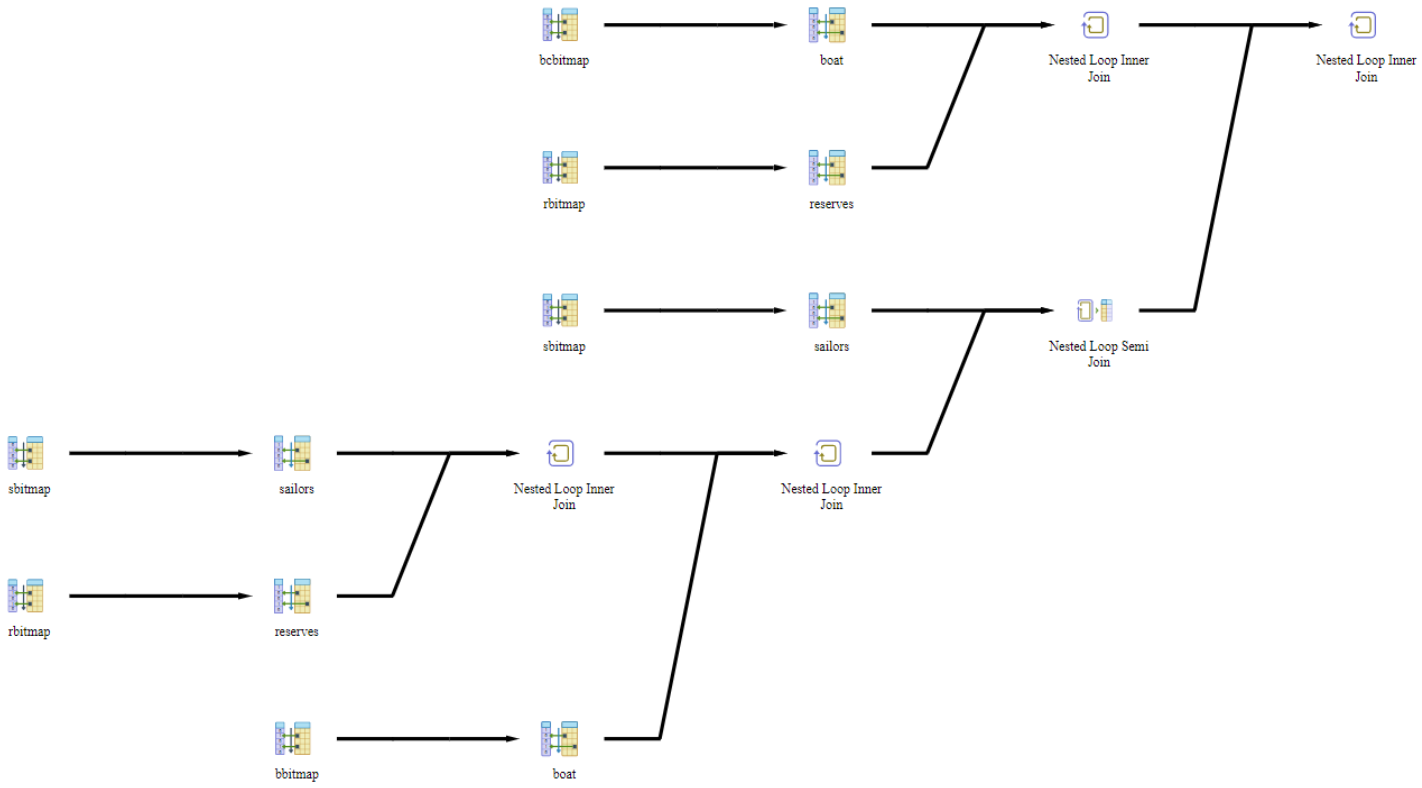
"Planning time: 14.008 ms"

"Execution time: 2285.359 ms"

1.Execution time1: 2285.359 ms"

2.Execution time2: 2126.677 ms

3.Execution time3: 2350.220 ms



Explanation:

Using bitmap index on tables boat, reserves & sailors where sbitmap is an index created on the primary key column of table sailors(sid), rbitmap is an index created on the primary key column of table reserves (sid,bid), bbitmapis an index created on the primary key column of table boat (bid)& bcbitmap is an index created on table boat using color column. Applying Bitmap Index Scan on sbitmap with index Cond: (sid = s.sid) then Bitmap Heap Scan on sailors s2 with Recheck Cond: (sid = s.sid) ,also applying Bitmap Index Scan on rbitmap with Index Cond: (sid = r.sid) then applying Bitmap Heap Scan on reserves r2 with Recheck Cond: (sid = r.sid), then applyin neestedloop inner join on both results from heap scan with Join Filter: (s2.sid = r2.sid), then joining the results with results of applying (Bitmap Index Scan on bbitmap with Index Cond: (bid = r2.bid) then applying Bitmap Heap Scan on boat b2 with Recheck Cond: (bid = r2.bid) and Filter: ((color)::text = 'Green'::text)") using neestedloop inner join then joining the results with applying (Bitmap Index Scan on sbitmap with Index Cond: (sid = r.sid) then applying Bitmap Heap Scan on sailors s with Recheck Cond: (sid = r.sid)) using neestedloop semi join then joining results with results of (joining between the results of applying (Bitmap Index Scan on rbitmap with Index Cond: (bid = b.bid) then applying Bitmap Heap Scan on reserves r with Recheck Cond: (bid = b.bid))and results of applying (Bitmap Index Scan on bcbitmap with Index Cond: ((color)::text = 'Red'::text) then applying Bitmap

Heap Scan on boat b with Recheck Cond: ((color)::text = 'Red'::text)) using neestedloop inner join) using neestedloop inner join.

So, the cost was estimated to be: 1555.26 with an average execution time after running query 3 times equal (2285.359 + 2126.677 + 2350.220)/3= 2254.085 ms. So, we notice that cost has decreased to compared to without index case due to using bitmap index, as it makes the operation faster so the execution time also decreased to 2254.085 ms so this execution time is accepted.

The best scenario:

	Without index	btree	hash	bitmap
Cost	67569.55	2226.02	334.71	1555.26
Exec.	222166.773	3693.140	567.619	2254.085

So, hash-based index is the best index for this query.

Schema4

Modifications on data insertion:

```
public static void populateMovie(Connection conn) {
for (int i = 1; i < 1000; i++) {

if (insertMovie(i, "movie" + i, i, i, "EN",
new Date((int) (Math.random() * 30 + 1), (int) (Math.random() * 12) + 1, 1999), "US", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateReviewer(Connection conn) {
for (int i = 1; i < 10000; i++) {

if (insertReviewer(i, "Name" + i, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateGenres(Connection conn) {

if (insertGenres(1, "Comedy", conn) == 0) {
System.err.println("insertion of record " + 1 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(2, "Romance", conn) == 0) {
System.err.println("insertion of record " + 2 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(3, "Drama", conn) == 0) {
System.err.println("insertion of record " + 3 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(4, "Action", conn) == 0) {
System.err.println("insertion of record " + 4 + " failed");
```

```
} else
System.out.println("insertion was successful");
if (insertGenres(5, "Historical", conn) == 0) {
System.err.println("insertion of record " + 5 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(6, "Tragedy", conn) == 0) {
System.err.println("insertion of record " + 6 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(7, "Sciencefiction", conn) == 0) {
System.err.println("insertion of record " + 7 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(8, "Documentaries", conn) == 0) {
System.err.println("insertion of record " + 8 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(9, "StoryOfLife", conn) == 0) {
System.err.println("insertion of record " + 9 + " failed");

} else
System.out.println("insertion was successful");
if (insertGenres(10, "LightMovie", conn) == 0) {
System.err.println("insertion of record " + 10 + " failed");

} else
System.out.println("insertion was successful");

}

public static void populateActor(Connection conn) {
for (int i = 1; i < 12000; i++) {
String result = "M";
if (i > 6000)
result = "F";
if (insertActor(i, "actor" + i, "actor" + i, result, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
```

```
}
}

public static void populateDirector(Connection conn) {
for (int i = 1; i < 2000; i++) {

if (insertDirector(i, "actor" + i, "actor" + i, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateMovieDirection(Connection conn) {
for (int i = 1; i < 1000; i++) {

if (insertMovieDirection(i, i, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateMovieCast(Connection conn) {
for (int i = 1; i < 1000; i++) {

if (insertMovieCast(i, i, "Princess", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(2 * i, i, "Prince", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(3 * i, i, "Doctor", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(4 * i, i, "Dancer", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
```

```
} else
System.out.println("insertion was successful");
if (insertMovieCast(5 * i, i, "Student", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(6 * i, i, "Teacher", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(7 * i, i, "Artist", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(8 * i, i, "Mother", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(9 * i, i, "Prisoner", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(10 * i, i, "Champion" , conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(11 * i, i, "Nurse", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertMovieCast(12 * i, i, "Manager", conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateMovieGenres(Connection conn) {
```

```
for (int i = 1; i < 1000; i++) {

if (insertMovieGenres(i, (int) Math.random() * 10 + 1, conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
}
}

public static void populateRating(Connection conn) {// accepted
for (int i = 1; i < 1000; i++) {

if (insertRating(i, i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 2 * i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");

if (insertRating(i,3* i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 4 * i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 5*i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 6 * i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 7*i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
```

```
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 8 * i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 9*i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");
if (insertRating(i, 10 * i, (int) (Math.random() * 5 + 1), (int) (Math.random() * 10 + 1), conn) == 0) {
System.err.println("insertion of record " + i + " failed");
break;
} else
System.out.println("insertion was successful");

}
}
```

Query10:

List all the information of the actors who played a role in the movie 'Annie Hall'.

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 ='movie1'));

results:

1	12	actor12	actor12	M
2	3	actor3	actor3	M
3	11	actor11	actor11	M
4	8	actor8	actor8	M
5	10	actor10	actor10	M
6	9	actor9	actor9	M
7	7	actor7	actor7	M
8	1	actor1	actor1	M
9	5	actor5	actor5	M
10	4	actor4	actor4	M
11	2	actor2	actor2	M
12	6	actor6	actor6	M

Without index:

Explain analyze output:

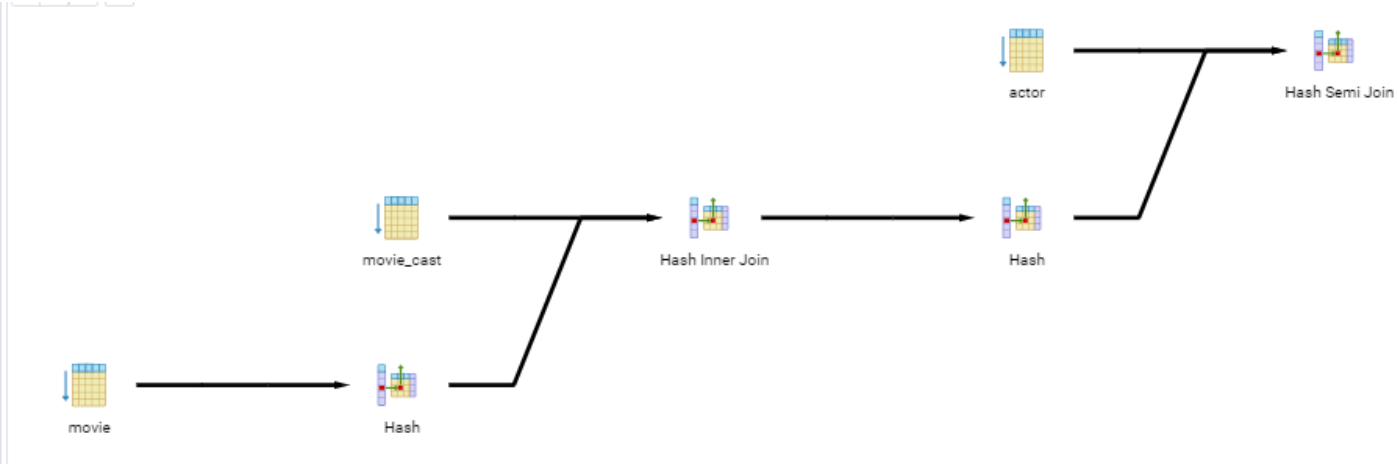
"Hash Semi Join (cost=284.13..548.75 rows=12 width=48) (actual time=9.206..16.986 rows=12 loops=1)"

"Execution Time: 17.042 ms"

"Execution Time: 20.453 ms"

"Execution Time: 17.813 ms"

	QUERY PLAN text
1	Hash Semi Join (cost=284.13..548.75 rows=12 width=48) (actual time=3.820..5.997 rows=12 loops=1)
2	Hash Cond: (actor.act_id = movie_cast.act_id)
3	-> Seq Scan on actor (cost=0.00..232.99 rows=11999 width=48) (actual time=0.020..1.102 rows=11999 loops=1)
4	-> Hash (cost=283.98..283.98 rows=12 width=4) (actual time=3.788..3.788 rows=12 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Join (cost=32.50..283.98 rows=12 width=4) (actual time=0.394..3.782 rows=12 loops=1)
7	Hash Cond: (movie_cast.mov_id = movie.mov_id)
8	-> Seq Scan on movie_cast (cost=0.00..219.88 rows=11988 width=8) (actual time=0.014..1.559 rows=11988 loops=1)
9	-> Hash (cost=32.49..32.49 rows=1 width=4) (actual time=0.372..0.372 rows=1 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on movie (cost=0.00..32.49 rows=1 width=4) (actual time=0.015..0.367 rows=1 loops=1)
12	Filter: (mov_title = 'movie1'::bpchar)
13	Rows Removed by Filter: 998
14	Planning Time: 0.433 ms
15	Execution Time: 6.043 ms



Explanation:

As we notice from query plan and explain analyze output that seqscan is used and no index is applied

Cost was found to be equal: 548.75 with an average execution time after running the query 3 times equal: (17.042 + 20.453 + 17.813)/3= 18.436 ms.

After bitmap index:

Flags: set enable_seqscan=off;

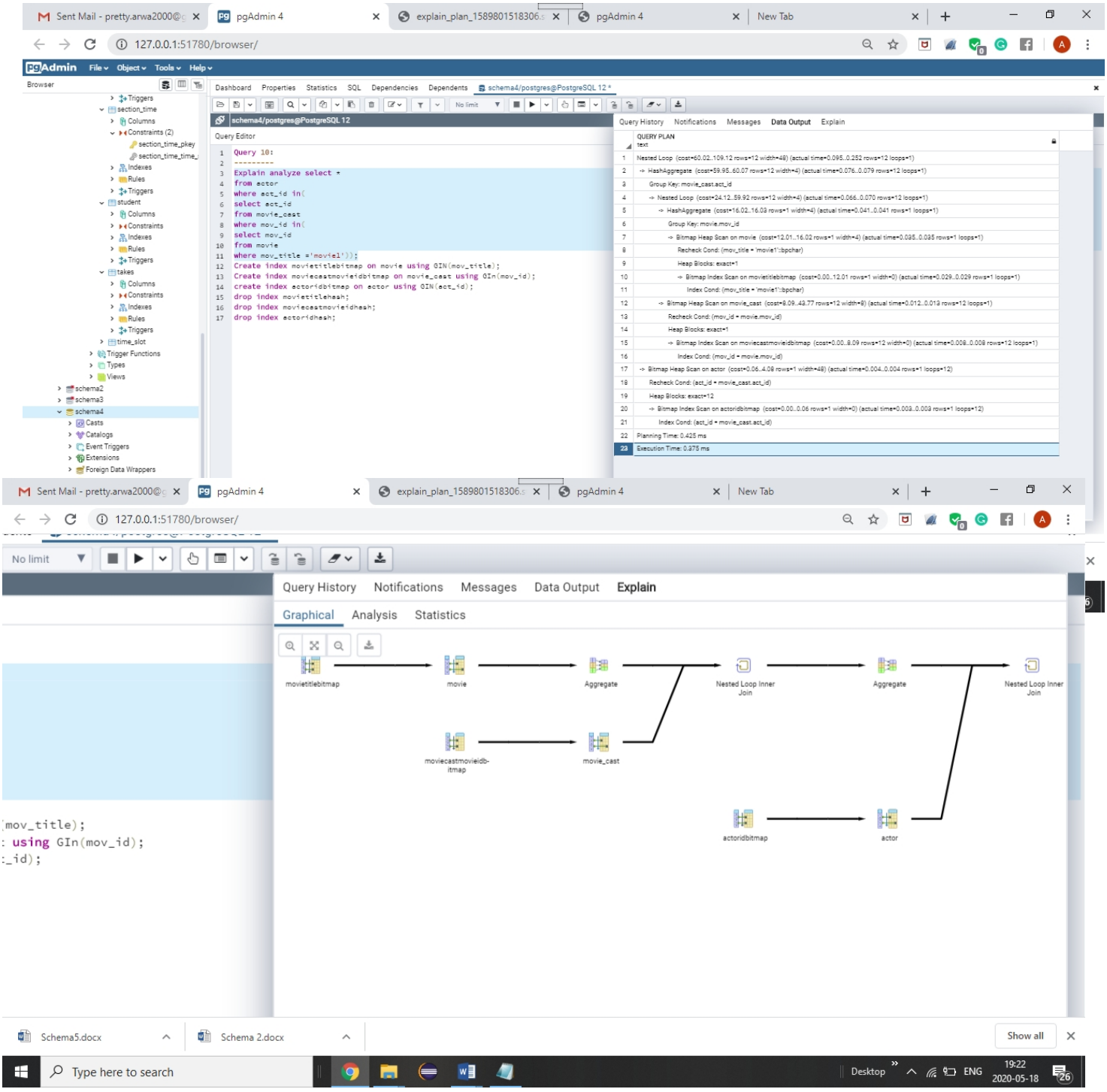
Explain analyze output:

"Nested Loop (cost=60.02..109.12 rows=12 width=48) (actual time=0.238..0.615 rows=12 loops=1)"

"Execution Time: 0.279 ms"

"Execution Time: 1.025 ms"

"Execution Time: 0.679 ms"



Explanation:

As we notice from explain analyze output and query plan that bitmap index is used

Cost was found to be equal: 109.12 with an average execution time after running the query 3 times equal: (0.279 + 1.025 + 0.679)/3= 0.661 ms. So, we notice that cost has decreased compared to without index case due to using bitmap index, as it makes the operation faster so the execution time also decreased to 0.661 ms so this execution time is accepted.

After hash index:

Flags: set enable_seqscan=off;

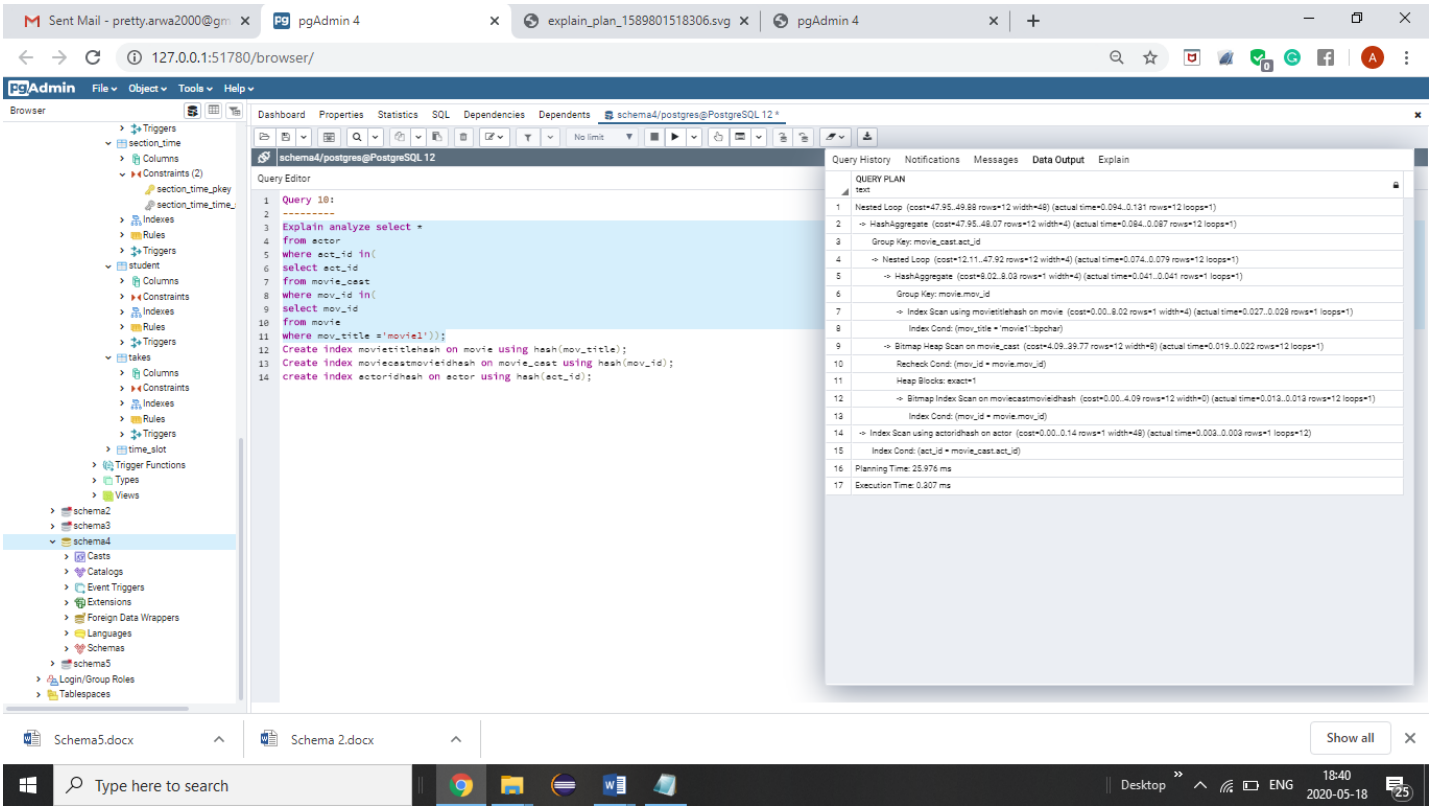
Explain analyze output:

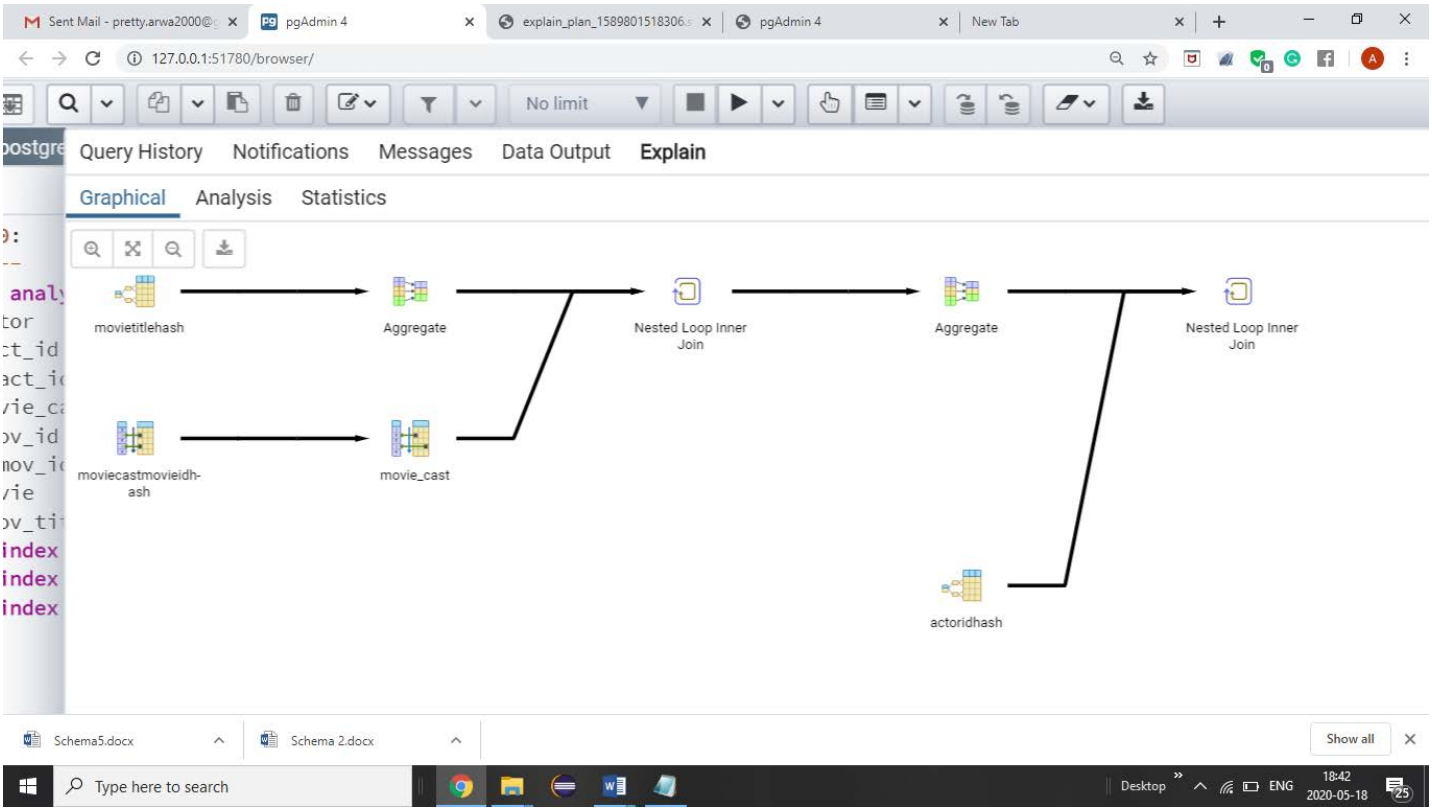
"Nested Loop (cost=47.95..49.88 rows=12 width=48) (actual time=0.094..0.131 rows=12 loops=1)"

"Execution Time: 0.307 ms"

"Execution Time: 0.623 ms"

"Execution Time: 0.115 ms"





Explanation:

As we notice from query plan and explain analyze output that hash index is used

Cost was found to be equal: 49.88 with an average execution time after running the query 3 times equal: (0.307 + 0.623 + 0.115)/3= 0.348 ms. So, we notice that cost has decreased compared to without index case due to using hash-based index, as hash complexity is O(1), while without index is O(n) so the execution time also decreased to 0.348 ms so this execution time is accepted

After btree index:

Flags: set enable_seqscan=off;

Explain analyze output:

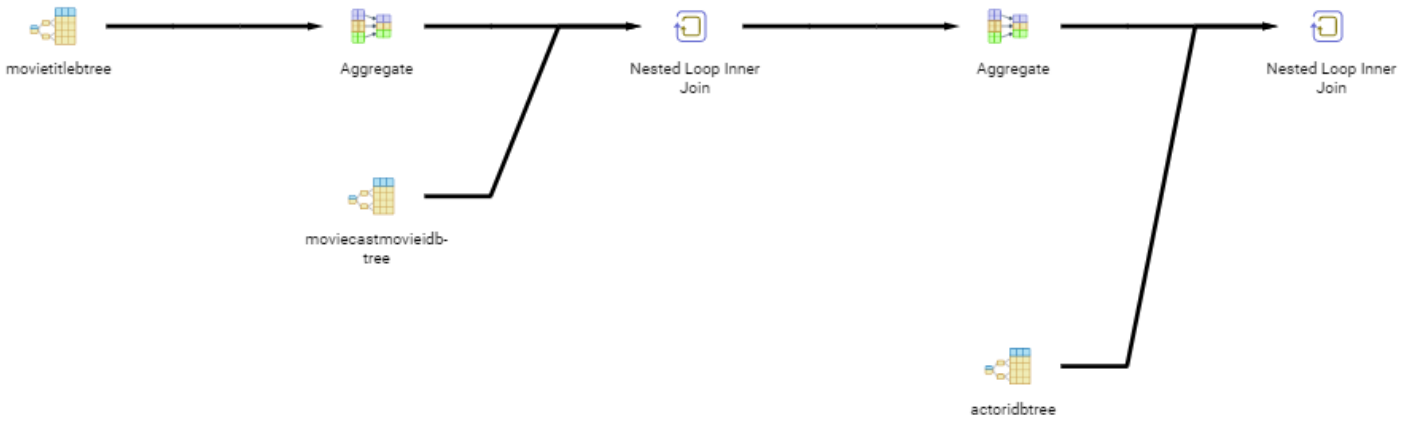
"Nested Loop (cost=17.23..22.04 rows=12 width=48) (actual time=0.119..0.144 rows=12 loops=1)"

"Execution Time: 0.094 ms"

"Execution Time: 0.122 ms"

"Execution Time: 0.142 ms"

	QUERY PLAN text
1	Nested Loop (cost=17.23..22.04 rows=12 width=48) (actual time=0.039..0.054 rows=12 loops=1)
2	-> HashAggregate (cost=16.95..17.07 rows=12 width=4) (actual time=0.036..0.039 rows=12 loops=1)
3	Group Key: movie_cast.act_id
4	-> Nested Loop (cost=8.58..16.92 rows=12 width=4) (actual time=0.032..0.034 rows=12 loops=1)
5	-> HashAggregate (cost=8.29..8.30 rows=1 width=4) (actual time=0.028..0.029 rows=1 loops=1)
6	Group Key: movie.mov_id
7	-> Index Scan using movietitlebtree on movie (cost=0.28..8.29 rows=1 width=4) (actual time=0.022..0.022 rows=1 loops=1)
8	Index Cond: (mov_title = 'movie1'::bpchar)
9	-> Index Scan using moviecastmovieidbtree on movie_cast (cost=0.29..8.49 rows=12 width=8) (actual time=0.003..0.004 rows=12 loop...
10	Index Cond: (mov_id = movie.mov_id)
11	-> Index Scan using actoridbtree on actor (cost=0.29..0.40 rows=1 width=48) (actual time=0.001..0.001 rows=1 loops=12)
12	Index Cond: (act_id = movie_cast.act_id)
13	Planning Time: 0.301 ms
14	Execution Time: 0.094 ms



Explanation:

As we notice from query plan and explain analyze output that btree index is used

Cost was found to be equal: 22.04 with an average execution time after running the query 3 times equal: (0.094 + 0.122 + 0.142)/3= 0.119 ms. so, we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 0.119 ms so this execution time is accepted.

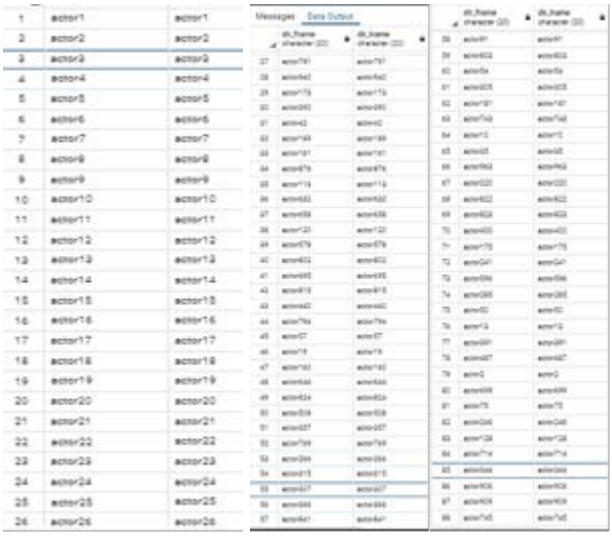
The best scenario:

	Without index	btree	hash	bitmap
Cost	548.75	22.04	49.88	109.12
Exec.	18.436	0.119	0.348	0.661

So, btree index is the best index for this query as it’s the least cost.

Query11:

Some of the results:



without index:

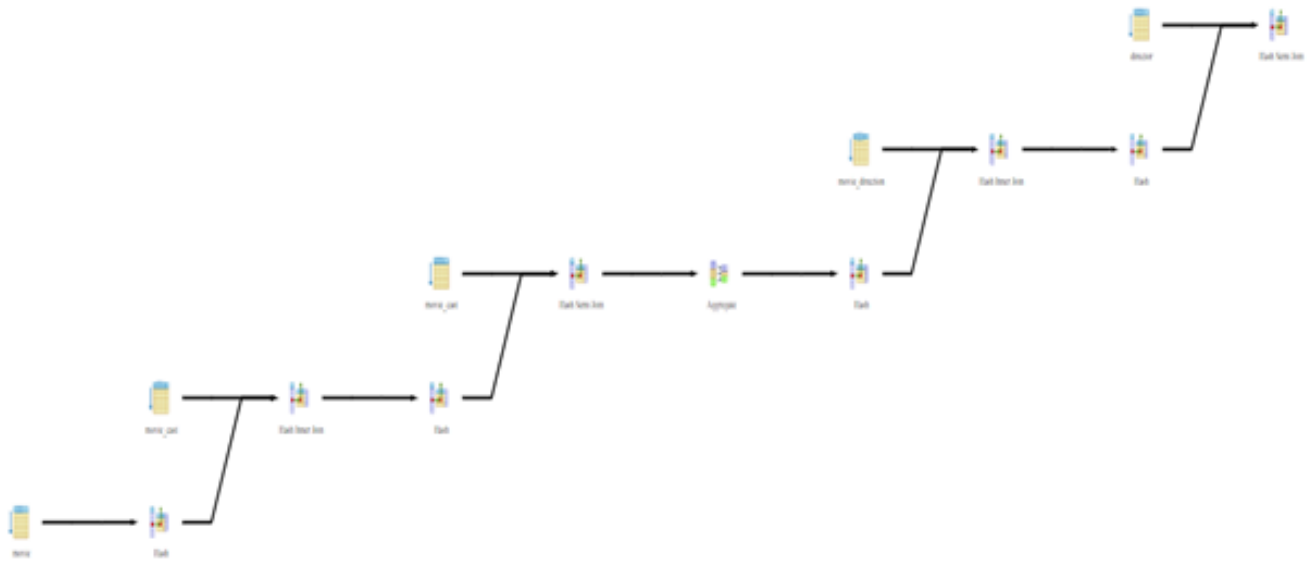
Explain analyze output:

"Hash Semi Join (cost=762.51..817.86 rows=999 width=42) (actual time=18.131..18.643 rows=999 loops=1)"

"Execution Time: 18.899 ms"

"Execution Time: 25.598 ms"

"Execution Time: 51.778 ms"



Explanation:

As we notice from query plan and explain analyze output that seqscan is used and no index is applied

Cost was found to be equal: 817.86 with an average execution time after running the query 3 times equal: (18.899 + 25.598 + 51.778)/3= 32.091 ms.

After bitmap index:

Flags: set enable_seqscan=off;

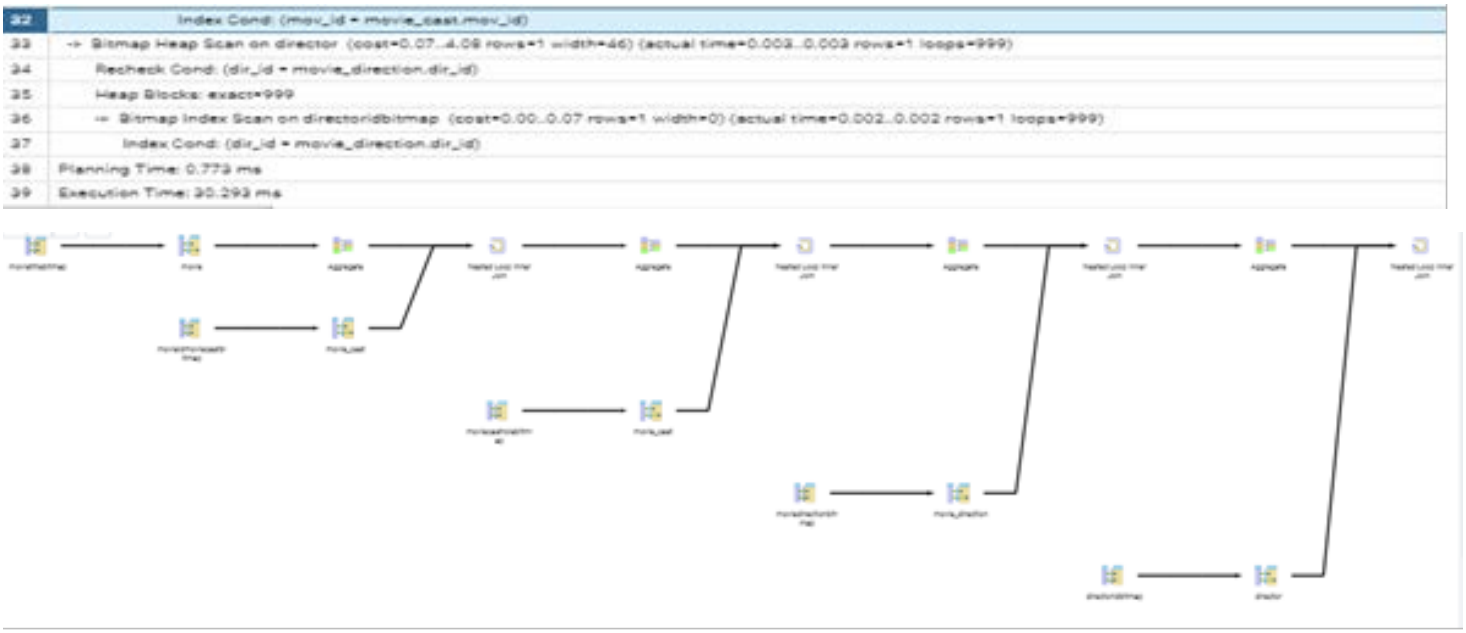
Explain analyze output:

"Nested Loop (cost=4874.26..8970.40 rows=999 width=42) (actual time=19.155..27.864 rows=999 loops=1)"

Execution Time: 30.293 ms

Execution Time: 30.122 ms

Execution Time: 40.114 ms



Explanation:

As we notice from query plan and explain analyze output that bitmap index is used

Cost was found to be equal: 8970.40 with an average execution time after running the query 3 times equal: (30.293 + 30.122 + 40.114)/3= 33.509 ms. So, we notice that cost has increased compared to without index case due to presence of duplicates and also eexecution time increased slightly

After btree index:

Flags: set enable_seqscan=off;

Explain analyze output:

"Merge Semi Join (cost=270.59..330.05 rows=999 width=42) (actual time=6.940..7.950 rows=999 loops=1)"

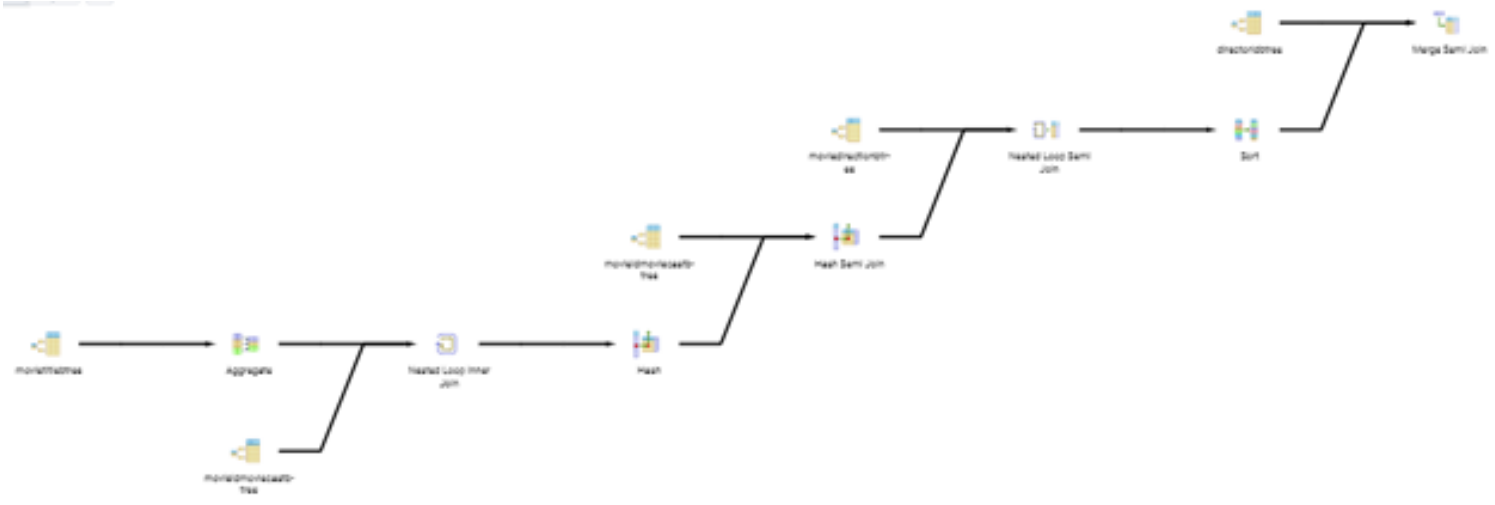
"Execution Time: 8.088 ms"

"Execution Time: 6.582 ms"

"Execution Time: 5.235 ms"

Messages | Data Output

	QUERY PLAN text
1	Merge Semi Join (cost=270.59..330.05 rows=999 width=42) (actual time=6.940..7.950 rows=999 loops=1)
2	Merge Cond: (director.dir_id = movie_direction.dir_id)
3	-> Index Scan using directoridbtree on director (cost=0.28..84.26 rows=1999 width=46) (actual time=0.010..0.255 rows=1000 loops=1)
4	-> Sort (cost=270.32..272.81 rows=999 width=4) (actual time=6.917..7.007 rows=999 loops=1)
5	Sort Key: movie_direction.dir_id
6	Sort Method: quicksort Memory: 71kB
7	-> Nested Loop Semi Join (cost=17.63..220.54 rows=999 width=4) (actual time=0.091..6.546 rows=999 loops=1)
8	-> Index Scan using moviedirectionbtree on movie_direction (cost=0.28..43.26 rows=999 width=8) (actual time=0.008..1.198 rows=999 loops=1)
9	-> Hash Semi Join (cost=17.35..19.27 rows=12 width=4) (actual time=0.005..0.005 rows=1 loops=999)
10	Hash Cond: (movie_cast.role = movie_cast_1.role)
11	-> Index Scan using movieidmoviecastbtree on movie_cast (cost=0.29..1.04 rows=12 width=35) (actual time=0.003..0.003 rows=1 loops=999)
12	Index Cond: (mov_id = movie_direction.mov_id)
13	-> Hash (cost=16.92..16.92 rows=12 width=31) (actual time=0.051..0.052 rows=12 loops=1)
14	Buckets: 1024 Batches: 1 Memory Usage: 9kB
15	-> Nested Loop (cost=8.58..16.92 rows=12 width=31) (actual time=0.039..0.044 rows=12 loops=1)
16	-> HashAggregate (cost=8.29..8.30 rows=1 width=4) (actual time=0.028..0.028 rows=1 loops=1)
17	Group Key: movie.mov_id
18	-> Index Scan using movietitlebtree on movie (cost=0.28..8.29 rows=1 width=4) (actual time=0.020..0.021 rows=1 loops=1)
19	Index Cond: (mov_title = 'movie2'::bpchar)
20	-> Index Scan using movieidmoviecastbtree on movie_cast movie_cast_1 (cost=0.29..8.49 rows=12 width=35) (actual time=0.006..0.009 rows=12 loops=1)
21	Index Cond: (mov_id = movie.mov_id)
22	Planning Time: 1.911 ms
23	Execution Time: 8.088 ms



Explanation:

As we notice from query plan and explain analyze output that btree index is used

Cost was found to be equal: 330.05 with an average execution time after running the query 3 times equal: (8.088 + 6.582 + 5.235)/3= 6.635 ms. so, we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 6.635 ms so this execution time is accepted.

After hash index:

Flags: set enable_seqscan=off;

Set enable_bitmapscan=off;

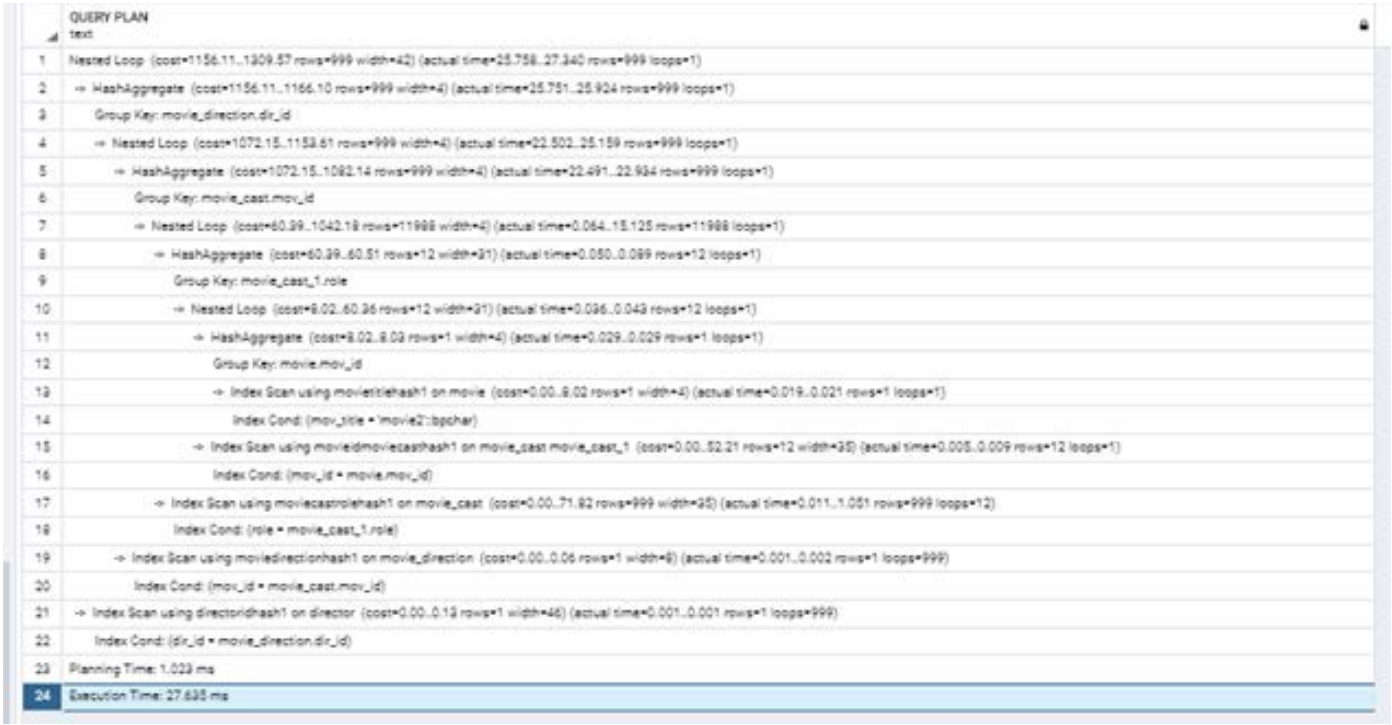
Explain analyze output:

"Nested Loop (cost=1156.11..1309.57 rows=999 width=42) (actual time=23.915..27.868 rows=999 loops=1)"

"Execution Time: 28.508 ms"

"Execution Time: 25.470 ms"

"Execution Time: 27.635 ms"



	QUERY PLAN text
1	Nested Loop (cost=1156.11..1309.57 rows=999 width=42) (actual time=25.758..27.840 rows=999 loops=1)
2	-> HashAggregate (cost=1156.11..1166.10 rows=999 width=4) (actual time=25.751..25.924 rows=999 loops=1)
3	Group Key: movie_direction.dir_id
4	-> Nested Loop (cost=1072.15..1153.61 rows=999 width=4) (actual time=22.502..25.159 rows=999 loops=1)
5	-> HashAggregate (cost=1072.15..1082.14 rows=999 width=4) (actual time=22.491..22.924 rows=999 loops=1)
6	Group Key: movie_cast.mov_id
7	-> Nested Loop (cost=60.99..1042.18 rows=11988 width=4) (actual time=0.064..15.125 rows=11988 loops=1)
8	-> HashAggregate (cost=60.99..60.51 rows=12 width=21) (actual time=0.050..0.089 rows=12 loops=1)
9	Group Key: movie_cast_1.role
10	-> Nested Loop (cost=8.02..60.96 rows=12 width=21) (actual time=0.036..0.049 rows=12 loops=1)
11	-> HashAggregate (cost=8.02..8.03 rows=1 width=4) (actual time=0.029..0.029 rows=1 loops=1)
12	Group Key: movie.mov_id
13	-> Index Scan using movietitlehash1 on movie (cost=0.00..8.02 rows=1 width=4) (actual time=0.019..0.021 rows=1 loops=1)
14	Index Cond: (mov_title = 'movie2'::bpchar)
15	-> Index Scan using movieidmoviecasthash1 on movie_cast movie_cast_1 (cost=0.00..52.21 rows=12 width=35) (actual time=0.005..0.009 rows=12 loops=1)
16	Index Cond: (mov_id = movie.mov_id)
17	-> Index Scan using moviecastrolehash1 on movie_cast (cost=0.00..71.82 rows=999 width=35) (actual time=0.011..1.051 rows=999 loops=12)
18	Index Cond: (role = movie_cast_1.role)
19	-> Index Scan using moviedirectionhash1 on movie_direction (cost=0.00..0.06 rows=1 width=8) (actual time=0.001..0.002 rows=1 loops=999)
20	Index Cond: (mov_id = movie_cast.mov_id)
21	-> Index Scan using directoridhash1 on director (cost=0.00..0.13 rows=1 width=46) (actual time=0.001..0.001 rows=1 loops=999)
22	Index Cond: (dir_id = movie_direction.dir_id)
23	Planning Time: 1.023 ms
24	Execution Time: 27.635 ms

Explanation:

As we notice from query plan and explain analyze output that hash index is used

Cost was found to be equal: 1309.57 with an average execution time after running the query 3 times equal: (28.508 + 25.470 + 27.635)/3= 27.204 ms. So, we notice that cost has increased compared to without index case due to presence of duplicates although hash complexity is O(1), while without index is O(n) so the execution time also decreased to 27.204 ms so this execution time is accepted.

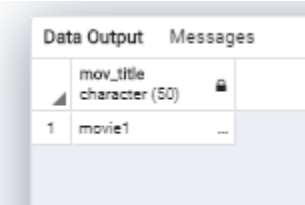
The best scenario:

	Without index	btree	hash	bitmap
Cost	817.86	330.05	1309.57	8970.40
Exec.	32.091	6.635	27.204	33.509

So, btree index is the best index for this query as it’s the least cost.

Query 12:

Results:



	mov_title character (50)
1	movie1

Without index:

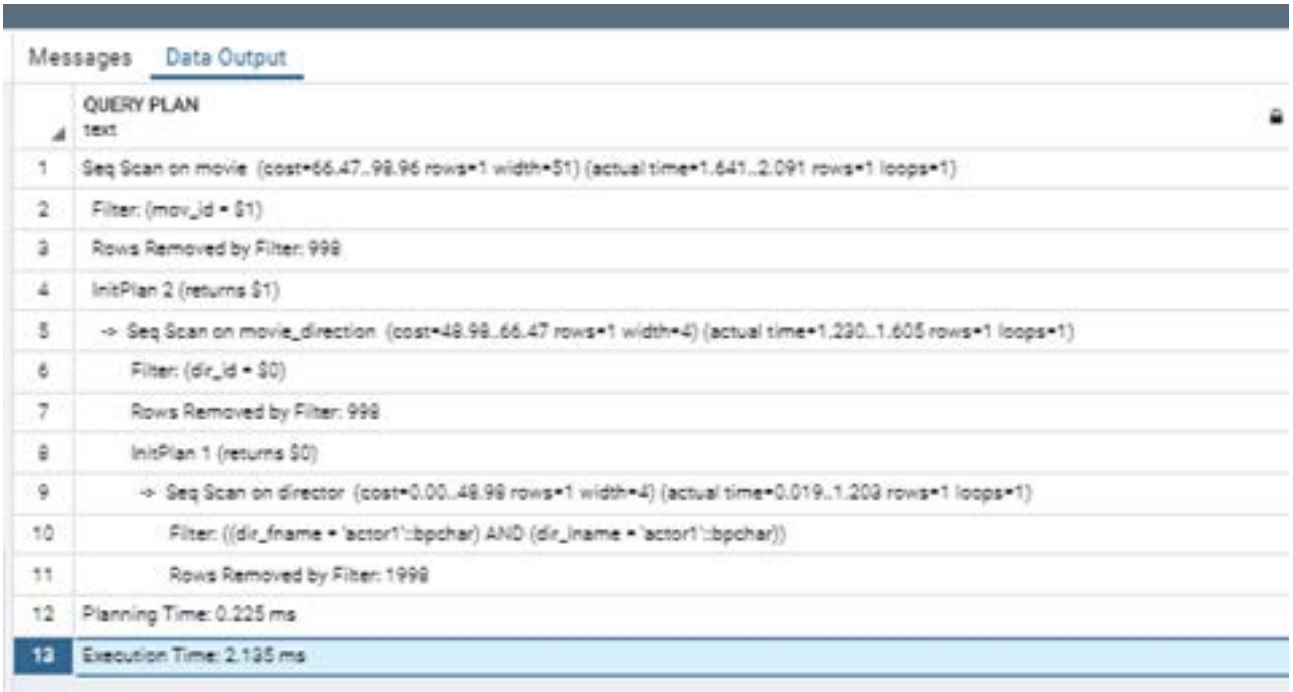
Explain analyze output:

"Seq Scan on movie (cost=66.47..98.96 rows=1 width=51) (actual time=0.659..0.886 rows=1 loops=1)"

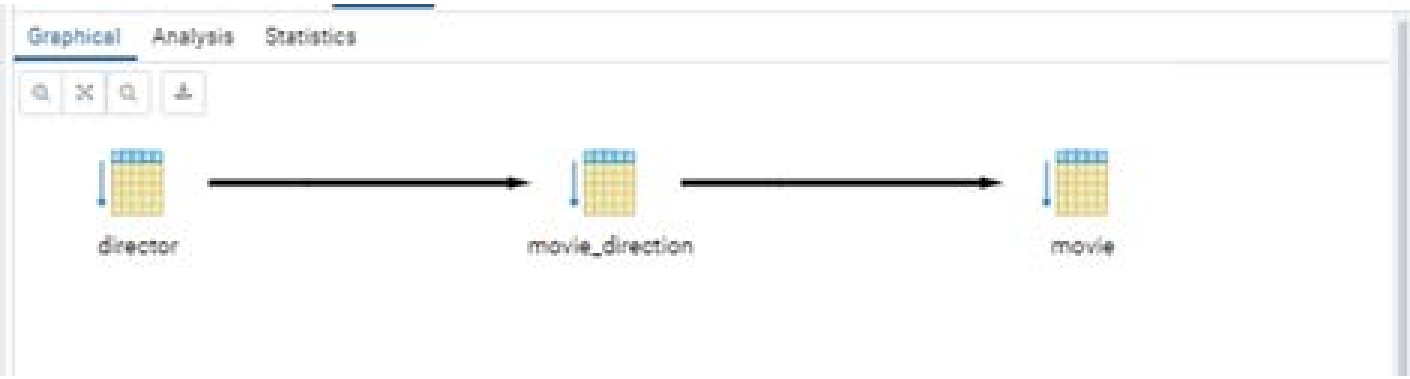
"Execution Time: 2.135 ms"

"Execution Time: 0.910 ms"

"Execution Time: 3.378 ms"



	QUERY PLAN text
1	Seq Scan on movie (cost=66.47..98.96 rows=1 width=51) (actual time=1.641..2.091 rows=1 loops=1)
2	Filter: (mov_id = $1)
3	Rows Removed by Filter: 999
4	InitPlan 2 (returns $1)
5	-> Seq Scan on movie_direction (cost=48.98..66.47 rows=1 width=4) (actual time=1.230..1.605 rows=1 loops=1)
6	Filter: (dir_id = $0)
7	Rows Removed by Filter: 999
8	InitPlan 1 (returns $0)
9	-> Seq Scan on director (cost=0.00..48.99 rows=1 width=4) (actual time=0.019..1.209 rows=1 loops=1)
10	Filter: ((dir_fname = 'actor1'::bpchar) AND (dir_lname = 'actor1'::bpchar))
11	Rows Removed by Filter: 1999
12	Planning Time: 0.225 ms
13	Execution Time: 2.135 ms



Explanation:

As we notice from query plan and explain analyze output that seqscan is used and no index is applied

Cost was found to be equal: 98.96 with an average execution time after running the query 3 times equal: (2.135 + 0.910 + 3.378)/3= 2.141 ms.

After bitmap index:

Flags: set enable_seqscan=off;

Explain analyze output:

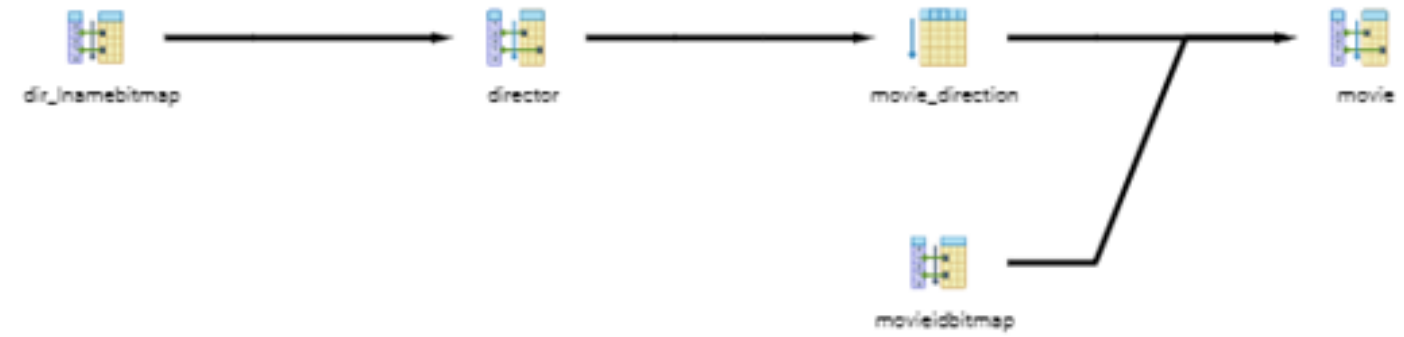
"Bitmap Heap Scan on movie (cost=41.52..45.53 rows=1 width=51) (actual time=0.882..0.883 rows=1 loops=1)"

"Execution Time: 1.136 ms"

"Execution Time: 0.591 ms"

"Execution Time: 0.619 ms"

	QUERY PLAN text
1	Bitmap Heap Scan on movie (cost=41.52..45.53 rows=1 width=51) (actual time=0.892..0.893 rows=1 loops=1)
2	Recheck Cond: (mov_id = $1)
3	Heap Blocks: exact=1
4	InitPlan 2 (returns $1)
5	-> Seq Scan on movie_direction (cost=16.02..33.51 rows=1 width=4) (actual time=0.192..0.840 rows=1 loops=1)
6	Filter: (dir_id = $0)
7	Rows Removed by Filter: 999
8	InitPlan 1 (returns $0)
9	-> Bitmap Heap Scan on director (cost=12.01..16.02 rows=1 width=4) (actual time=0.125..0.126 rows=1 loops=1)
10	Recheck Cond: (dir_lname = 'actor1'::bpchar)
11	Filter: (dir_fname = 'actor1'::bpchar)
12	Heap Blocks: exact=1
13	-> Bitmap Index Scan on dir_lnamebitmap (cost=0.00..12.01 rows=1 width=0) (actual time=0.100..0.100 rows=...
14	Index Cond: (dir_lname = 'actor1'::bpchar)
15	-> Bitmap Index Scan on movieidbitmap (cost=0.00..8.01 rows=1 width=0) (actual time=0.871..0.871 rows=1 loops=1)
16	Index Cond: (mov_id = $1)
17	Planning Time: 1.254 ms
18	Execution Time: 1.136 ms



Explanation:

As we notice from explain analyze output that bitmap index is used

Cost was found to be equal: 45.53 with an average execution time after running the query 3 times equal: (1.136 + 0.591 + 0.619)/3= 0.782 ms. So, we notice that cost has decreased compared to without index case due to using bitmap index, as it

makes the operation faster so the execution time also decreased to 0.782 ms so this execution time is accepted.

After hash index:

Flags: set enable_seqscan=off;

Explain analyze output:

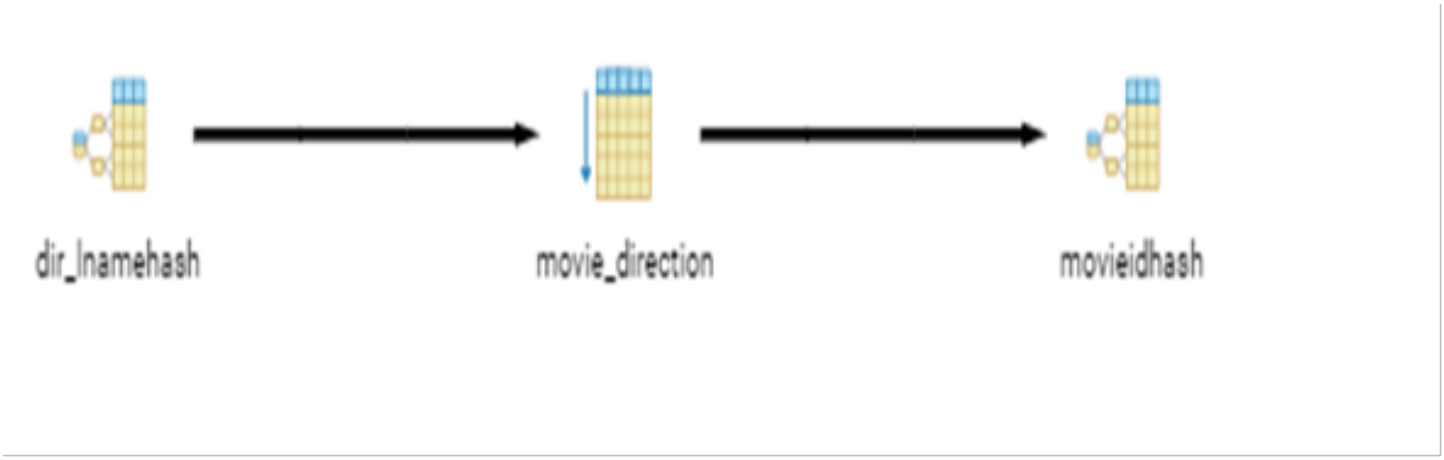
"Index Scan using movieidhash on movie (cost=25.51..33.52 rows=1 width=51) (actual time=0.396..0.397 rows=1 loops=1)""Execution Time: 0.148 ms"

"Execution Time: 0.458 ms"

"Execution Time: 0.596 ms"

"Execution Time: 0.293 ms"

	QUERY PLAN text
1	Index Scan using movieidhash on movie (cost=25.51..33.52 rows=1 width=51) (actual time=0.527..0.529 rows=1 loops=1)
2	Index Cond: (mov_id = $1)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on movie_direction (cost=9.02..25.51 rows=1 width=4) (actual time=0.078..0.498 rows=1 loops=1)
5	Filter: (dir_id = $0)
6	Rows Removed by Filter: 999
7	InitPlan 1 (returns $0)
8	-> Index Scan using dir_lnamehash on director (cost=0.00..9.02 rows=1 width=4) (actual time=0.024..0.025 rows=1 loops=1)
9	Index Cond: (dir_lname = 'actor1'::bpchar)
10	Filter: (dir_fname = 'actor1'::bpchar)
11	Planning Time: 0.662 ms
12	Execution Time: 0.596 ms



Explanation:

As we notice from query plan and explain analyze output that hash index is used

Cost was found to be equal: 33.52 with an average execution time after running the query 3 times equal: (0.458 + 0.596 + 0.293)/3= 0.449 ms. So, we notice that cost has decreased compared to without index case due to using hash-based index, as hash complexity is O(1), while without index is O(n) so the execution time also decreased to 0.449 ms so this execution time is accepted

After btree index:

Flags: set enable_seqscan=off;

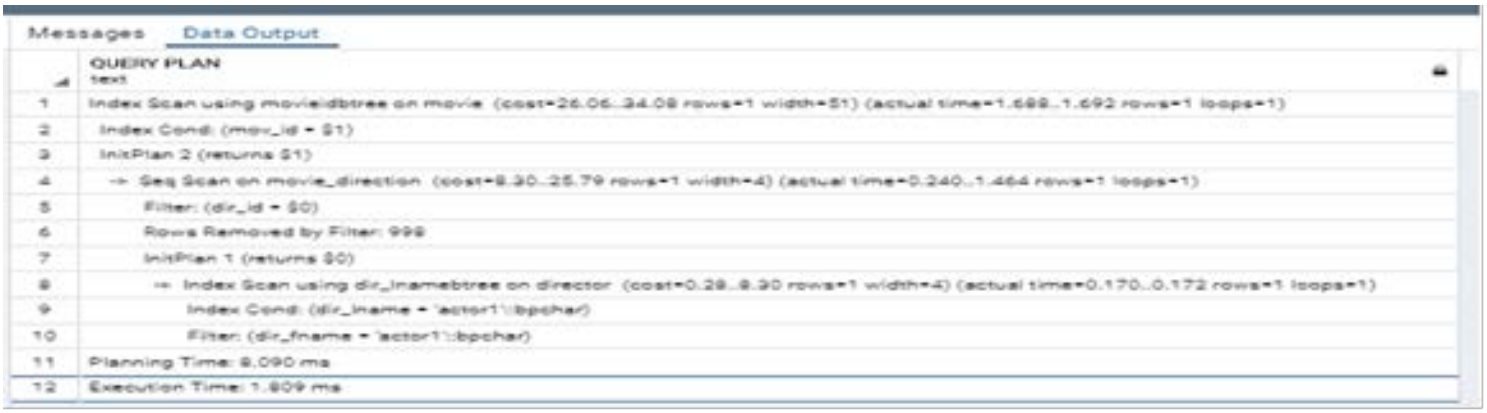
Explain analyze output:

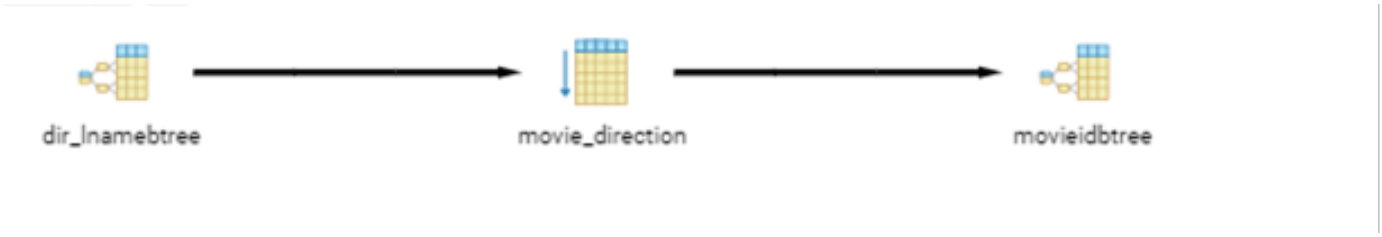
"Index Scan using movieidbtree on movie (cost=26.06..34.08 rows=1 width=51) (actual time=1.688..1.692 rows=1 loops=1)""Execution Time: 0.121 ms"

"Execution Time: 1.809 ms"

"Execution Time: 1.083 ms"

"Execution Time: 0.373 ms"





As we notice from query plan and explain analyze output that btree index is used

Cost was found to be equal: 34.08 with an average execution time after running the query 3 times equal: (1.809 + 1.083 + 0.373)/3= 1.088 ms. so, we notice that cost has decreased compared to without index case due to using btree index, as btree complexity is O(logn) while without index is O(n) so the execution time also decreased to 1.088 ms so this execution time is accepted.

The best scenario:

	Without index	btree	hash	bitmap
Cost	98.96	34.08	33.52	45.53
Exec.	2.141	1.088	0.449	0.782

So, hash index is the best index for this query as it's the least cost.

Explanation:

As we notice from query plan and explain analyze output that hash index is used

Cost was found to be equal: 1264.90 with an average execution time after running the query 3 times equal: (16.160 + 15.540 + 19.717)/3= 17.139 ms. So, we notice that cost has increased compared to without index case due to presence of duplicates although hash complexity is O(1), while without index is O(n) so the execution time also decreased to 17.139 ms so this execution time is accepted.

The best scenario:

	Without index	btree	hash	bitmap
Cost	817.86	330.05	1264.90	8970.40
Exec.	32.091	6.635	17.139	33.509

So, btree index is the best index for this query as it’s the least cost.

Schema 5

1. Modifications on Data Insertions

As required in the project description, that number of countries should be 195, the number of matches should be 5000, and the number of players should be 10000. Hence, to make sure that the data is consistent, and the results of the query will not be empty; I changed the following in the data population code:

1. *playing_position: I changed it to 9 positions to be more realistic according to the number of existing positions in soccer.*
2. *soccer_country: I changed to 195 as required in the project description.*
3. *player_mast: No changes in the number of players in the original insertion code, but I made sure to respect the constraints of the foreign keys by using the modulus operator '%'.*
4. *coach_mast: I changed the number of coaches to 195, assuming that there is only a coach per team.*
5. *soccer_city: I changed it to be 390, so every country will have two soccer cities.*
6. *soccer_venue: I changed it to be 780 venues, that means every soccer city will have two soccer venues.*
7. *refree_mast: I changed it to be 858 referees (3 referees per team).*
8. *asst_referee_mast: I changed it to be 1170 (6 assistant referees per team).*
9. *match_mast: I changed to 5000 as required in the project description, but I made sure to respect the constraints of the foreign keys by using the modulus operator '%'.*
10. *match_details: I changed to be 9998, since the first two matches will have only one team (Germany1 and Germany2) for the queries 14, 15 will not give empty results since there is in the queries (Select match_no from match_details where team_id in… group by match_no having count(distinct team_id) = 1) , so that means we want matches that having only one team and since in the queries (team_id in (select country_id where country_name = 'Germany1') or (select country_id where country_name = 'Germany2')) that means that teams 'Germany1' and 'Germany2' at least should be included in that assumption. Also, for query 17 to work, 'Germany1' and 'Germany2' should have played only one match together, since the sub-queries in query 17 started with '=', and that means the results count should be only one. I also made sure not to violate any foreign key constraints by the modulus operator '%'.*
11. *team_coaches: I changed to 195, where every team will have one coach.*

12. *soccer_team: I changed to 195, where every country will have only one soccer team.*
13. *player_in_out: I changed to total of 5000, respecting the foreign key constraints.*
14. *goal_details: I changed to be 5000 goals, a goal per match and the match between 'Germany1' and 'Germany2' has only one goal for 'Germany2' by player 2 and his name is 'Salah2' to make sure that query 17 will not have an empty result.*
15. *penalty_shootout: I changed to total of 5000, respecting the foreign key constraints.*
16. *match_captain: I changed to total of 9998, respecting the foreign key constraints, where there is a captain per team in all of the matches.*
17. *penalty_gk: I changed to total of 5000, respecting the foreign key constraints, where there is a goal keeper for every happening penalty.*
18. *Player_booked: I changed to total of 5000, respecting the foreign key constraints, where there is one player booked per match.*

2. *Query 13*

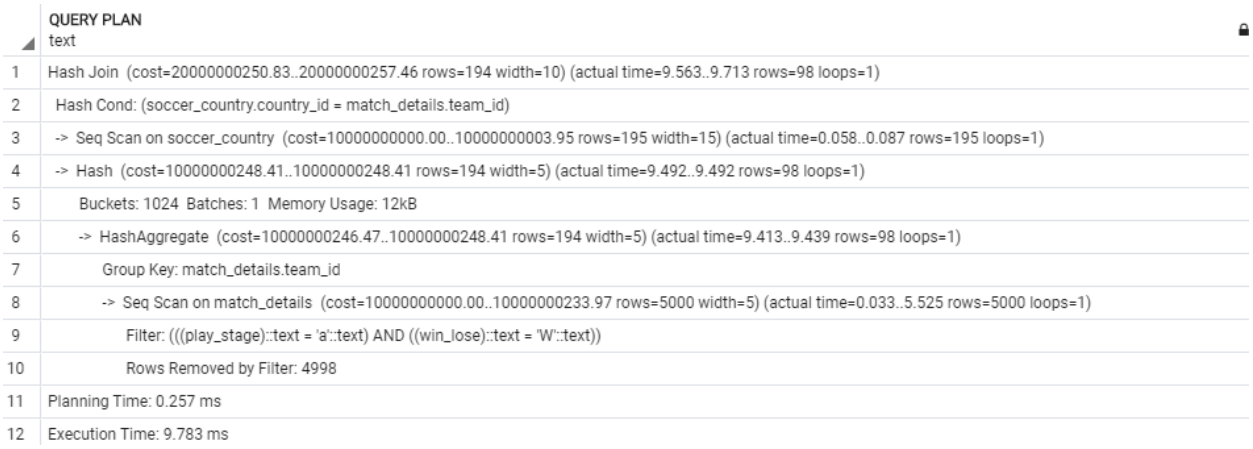
a. *Index on match_details.play_stage:*

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Without Indices:*
  - *Query Plan: Figure (i).*
  - *Estimated Cost: 20000000257.46.*
  - *Average Execution Time: 5.16ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 1.a*
  - *Estimated Cost: 10000000430.73.*
  - *Average Execution Time: 5.525ms.*
  - *Explanation: The cost has decreased due to the usage of the B-tree index that has time complexity of O (log n) instead of the linear searching of O (n). But the execution time has slightly increased since that the data in this column has only one value, which is used in the query. Hence, it has to do linear searching, which gives the same complexity as the case that as if an index was not applied.*
- *Hash-based Index:*
  - *Query Plan: figure 1.b*
  - *Estimated Cost: 10000000693.69*
  - *Average Execution Time: 5.69ms.*
  - *Explanation: The cost has decreased since we used hash-based index of time complexity O (1) but the execution time has increased due to the many duplicates in this column.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 1.c.*

- *Estimated Cost: 10000000353.69*
- *Average Execution Time: 5.482ms*
- *Explanation: The cost has decreased since we used bitmap index but the execution time has increased due to the many duplicates in this column which gives time complexity of O (n) since the bitmap line will have a line of 1s.*

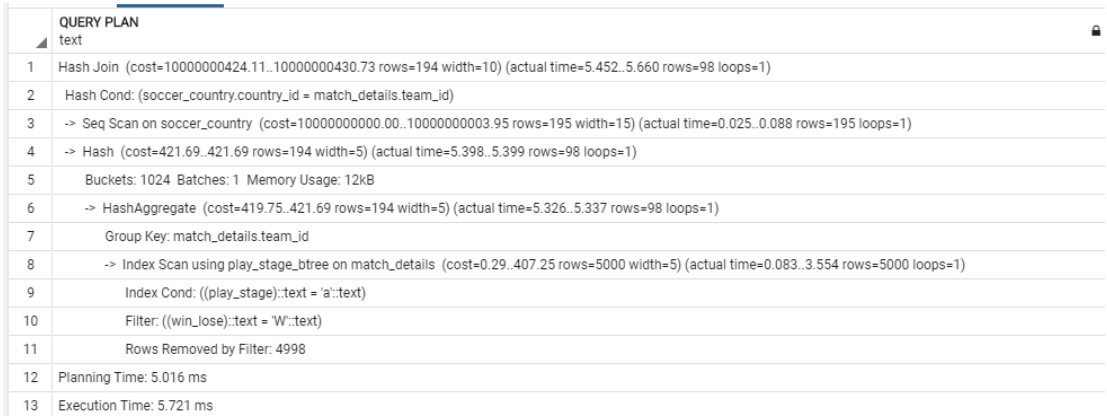
- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is not using any indices on this column, since it has the least execution time.



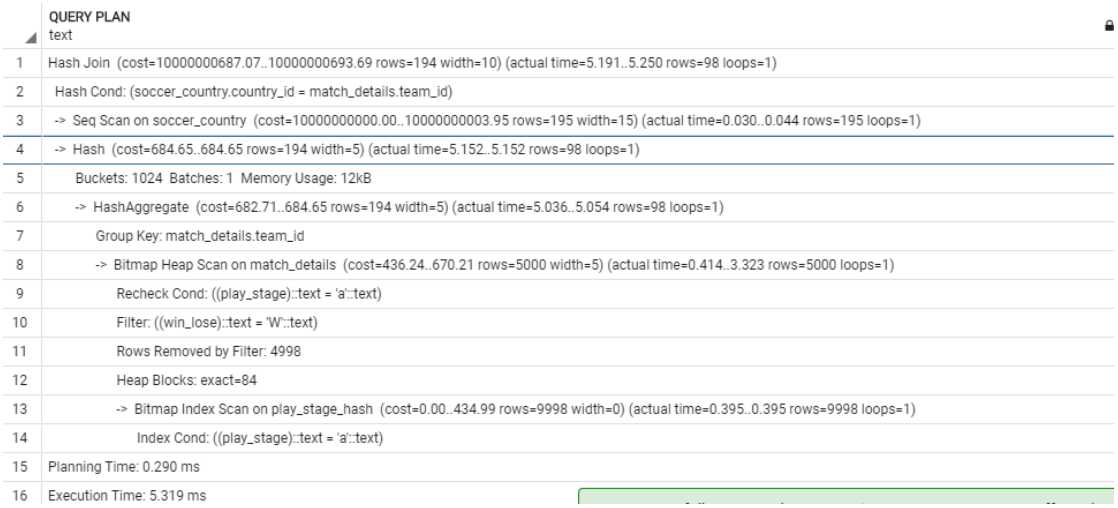
	QUERY PLAN text
1	Hash Join (cost=20000000250.83..20000000257.46 rows=194 width=10) (actual time=9.563..9.713 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.058..0.087 rows=195 loops=1)
4	-> Hash (cost=10000000248.41..10000000248.41 rows=194 width=5) (actual time=9.492..9.492 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=10000000246.47..10000000248.41 rows=194 width=5) (actual time=9.413..9.439 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=5000 width=5) (actual time=0.033..5.525 rows=5000 loops=1)
9	Filter: (((play_stage)::text = 'a'::text) AND ((win_lose)::text = 'W'::text))
10	Rows Removed by Filter: 4998
11	Planning Time: 0.257 ms
12	Execution Time: 9.783 ms

Figure (i)



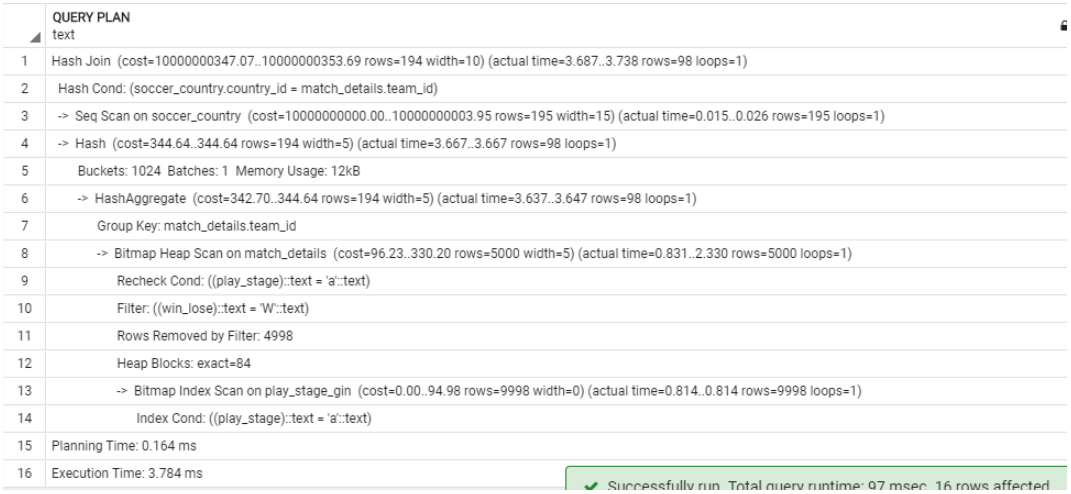
	QUERY PLAN text
1	Hash Join (cost=10000000424.11..10000000430.73 rows=194 width=10) (actual time=5.452..5.660 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.025..0.088 rows=195 loops=1)
4	-> Hash (cost=421.69..421.69 rows=194 width=5) (actual time=5.398..5.399 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=419.75..421.69 rows=194 width=5) (actual time=5.326..5.337 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Index Scan using play_stage_btree on match_details (cost=0.29..407.25 rows=5000 width=5) (actual time=0.083..3.554 rows=5000 loops=1)
9	Index Cond: ((play_stage)::text = 'a'::text)
10	Filter: ((win_lose)::text = 'W'::text)
11	Rows Removed by Filter: 4998
12	Planning Time: 5.016 ms
13	Execution Time: 5.721 ms

Figure 1.a



	QUERY PLAN text
1	Hash Join (cost=10000000687.07..10000000693.69 rows=194 width=10) (actual time=5.191..5.250 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.030..0.044 rows=195 loops=1)
4	-> Hash (cost=684.65..684.65 rows=194 width=5) (actual time=5.152..5.152 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=682.71..684.65 rows=194 width=5) (actual time=5.036..5.054 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Bitmap Heap Scan on match_details (cost=436.24..670.21 rows=5000 width=5) (actual time=0.414..3.323 rows=5000 loops=1)
9	Recheck Cond: ((play_stage)::text = 'a'::text)
10	Filter: ((win_lose)::text = 'W'::text)
11	Rows Removed by Filter: 4998
12	Heap Blocks: exact=84
13	-> Bitmap Index Scan on play_stage_hash (cost=0.00..434.99 rows=9998 width=0) (actual time=0.395..0.395 rows=9998 loops=1)
14	Index Cond: ((play_stage)::text = 'a'::text)
15	Planning Time: 0.290 ms
16	Execution Time: 5.319 ms

Figure 1.b



	QUERY PLAN text
1	Hash Join (cost=10000000347.07..10000000353.69 rows=194 width=10) (actual time=3.687..3.738 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.015..0.026 rows=195 loops=1)
4	-> Hash (cost=344.64..344.64 rows=194 width=5) (actual time=3.667..3.667 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=342.70..344.64 rows=194 width=5) (actual time=3.637..3.647 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Bitmap Heap Scan on match_details (cost=96.23..330.20 rows=5000 width=5) (actual time=0.831..2.330 rows=5000 loops=1)
9	Recheck Cond: ((play_stage)::text = 'a'::text)
10	Filter: ((win_lose)::text = 'W'::text)
11	Rows Removed by Filter: 4998
12	Heap Blocks: exact=84
13	-> Bitmap Index Scan on play_stage_gin (cost=0.00..94.98 rows=9998 width=0) (actual time=0.814..0.814 rows=9998 loops=1)
14	Index Cond: ((play_stage)::text = 'a'::text)
15	Planning Time: 0.164 ms
16	Execution Time: 3.784 ms

Figure 1.c

b. Index on match_details.win_lose:

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Without Indices:*
  - *Query Plan: Figure (i).*
  - *Estimated Cost: 20000000257.46.*
  - *Average Execution Time: 5.16ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 2.a*

- *Estimated Cost: 10000000285.52.*
- *Average Execution Time: 4.422ms.*
- *Explanation: The cost has decreased due to the usage of the B-tree index that has time complexity of O (log n) instead of the linear searching of O (n), and we can see that it affects the execution time.*

- *Hash-based Index:*
  - *Query Plan: figure 2.b*
  - *Estimated Cost: 10000000405.24.*
  - *Average Execution Time: 4.69ms.*
  - *Explanation: The cost has decreased since we used hash-based index of time complexity O (1).*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 2.c.*
  - *Estimated Cost: 10000000233.24.*
  - *Average Execution Time: 3.736ms*
  - *Explanation: The cost has decreased since we used bitmap index that uses bit-wise operations to filter the columns, which is really fast.*
- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the GIN index on this column, since it has the least execution time, and it is also very fast with AND operation.

	QUERY PLAN text
1	Hash Join (cost=10000000278.90..10000000285.52 rows=194 width=10) (actual time=5.158..5.292 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.034..0.055 rows=195 loops=1)
4	-> Hash (cost=276.47..276.47 rows=194 width=5) (actual time=5.116..5.117 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=274.53..276.47 rows=194 width=5) (actual time=5.058..5.076 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Bitmap Heap Scan on match_details (cost=103.03..262.03 rows=5000 width=5) (actual time=0.552..2.558 rows=5000 loops=1)
9	Recheck Cond: ((win_lose)::text = 'W'::text)
10	Filter: ((play_stage)::text = 'a'::text)
11	Heap Blocks: exact=84
12	-> Bitmap Index Scan on win_lose_btree (cost=0.00..101.78 rows=5000 width=0) (actual time=0.536..0.536 rows=5000 loops=1)
13	Index Cond: ((win_lose)::text = 'W'::text)
14	Planning Time: 1.369 ms
15	Execution Time: 5.350 ms

Figure 2.a

	QUERY PLAN text
1	Hash Join (cost=10000000398.61..10000000405.24 rows=194 width=10) (actual time=5.172..5.299 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.039..0.063 rows=195 loops=1)
4	-> Hash (cost=396.19..396.19 rows=194 width=5) (actual time=5.124..5.125 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=394.25..396.19 rows=194 width=5) (actual time=5.045..5.066 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Bitmap Heap Scan on match_details (cost=222.75..381.75 rows=5000 width=5) (actual time=0.229..2.553 rows=5000 loops=1)
9	Recheck Cond: ((win_lose)::text = 'W'::text)
10	Filter: ((play_stage)::text = 'a'::text)
11	Heap Blocks: exact=84
12	-> Bitmap Index Scan on win_lose_hash (cost=0.00..221.50 rows=5000 width=0) (actual time=0.211..0.211 rows=5000 loops=1)
13	Index Cond: ((win_lose)::text = 'W'::text)
14	Planning Time: 1.713 ms
15	Execution Time: 5.375 ms

Figure 2.b

Query Editor | Data Output | Messages | Explain | Notifications | Query History

	QUERY PLAN text
1	Hash Join (cost=10000000226.61..10000000233.24 rows=194 width=10) (actual time=2.989..3.038 rows=98 loops=1)
2	Hash Cond: (soccer_country.country_id = match_details.team_id)
3	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.016..0.025 rows=195 loops=1)
4	-> Hash (cost=224.19..224.19 rows=194 width=5) (actual time=2.963..2.963 rows=98 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> HashAggregate (cost=222.25..224.19 rows=194 width=5) (actual time=2.935..2.945 rows=98 loops=1)
7	Group Key: match_details.team_id
8	-> Bitmap Heap Scan on match_details (cost=50.75..209.75 rows=5000 width=5) (actual time=0.435..1.431 rows=5000 loops=1)
9	Recheck Cond: ((win_lose)::text = 'W'::text)
10	Filter: ((play_stage)::text = 'a'::text)
11	Heap Blocks: exact=84
12	-> Bitmap Index Scan on win_lose_gin (cost=0.00..49.50 rows=5000 width=0) (actual time=0.420..0.420 rows=5000 loops=1)
13	Index Cond: ((win_lose)::text = 'W'::text)
14	Planning Time: 2.133 ms
15	Execution Time: 3.096 ms

Figure 2.c

c. *Index on soccer_country.country_id*

- *Flags Modifications: The flags enable_seqscan, enable_hashjoin, and enable_mergejoin were set off.*
- *Without Indices:*
  - *Query Plan: Figure (ii).*
  - *Estimated Cost: 20000000820.3.*
  - *Average Execution Time: 11.523ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 3.a*
  - *Estimated Cost: 10000000297.88.*
  - *Average Execution Time: 5.39ms.*
  - *Explanation: The cost has decreased due to the usage of the B-tree index that has time complexity of O (log n) instead of the linear searching of O (n), and we can see that it affects in the execution time.*
- *Hash-based Index:*
  - *Query Plan: Figure 3.b*
  - *Estimated Cost: 10000000277.74.*
  - *Average Execution Time: 4.713ms.*
  - *Explanation: The cost has decreased since we used hash-based index of time complexity O (1).*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 3.c.*
  - *Estimated Cost: 10000001038.28.*
  - *Average Execution Time: 5.417ms*

- *Explanation: The cost has decreased since we used bitmap index that uses bit-wise operations to filter the columns, which is really fast.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index due to its constant time complexity.

	QUERY PLAN text
1	Nested Loop (cost=20000000246.47..20000000820.30 rows=194 width=10) (actual time=4.959..14.723 rows=98 loops=1)
2	Join Filter: (soccer_country.country_id = match_details.team_id)
3	Rows Removed by Join Filter: 19012
4	-> HashAggregate (cost=10000000246.47..10000000248.41 rows=194 width=5) (actual time=4.854..4.909 rows=98 loops=1)
5	Group Key: match_details.team_id
6	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=5000 width=5) (actual time=0.025..2.775 rows=5000 loops=1)
7	Filter: (((play_stage)::text = 'a'::text) AND ((win_lose)::text = 'W'::text))
8	Rows Removed by Filter: 4998
9	-> Materialize (cost=10000000000.00..10000000004.93 rows=195 width=15) (actual time=0.000..0.018 rows=195 loops=98)
10	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.019..0.046 rows=195 loops=1)
11	Planning Time: 0.131 ms
12	Execution Time: 14.769 ms

Figure (ii)

Query Editor | Data Output | Messages | Explain | Notifications | Query History

	QUERY PLAN text
1	Nested Loop (cost=10000000246.61..10000000297.88 rows=194 width=10) (actual time=6.056..6.321 rows=98 loops=1)
2	-> HashAggregate (cost=10000000246.47..10000000248.41 rows=194 width=5) (actual time=6.027..6.043 rows=98 loops=1)
3	Group Key: match_details.team_id
4	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=5000 width=5) (actual time=0.030..3.557 rows=5000 loops=1)
5	Filter: (((play_stage)::text = 'a'::text) AND ((win_lose)::text = 'W'::text))
6	Rows Removed by Filter: 4998
7	-> Index Scan using country_id_btree on soccer_country (cost=0.14..0.24 rows=1 width=15) (actual time=0.002..0.002 rows=1 loops=98)
8	Index Cond: (country_id = match_details.team_id)
9	Planning Time: 0.210 ms
10	Execution Time: 6.381 ms

Figure 3.a

	QUERY PLAN text
1	Nested Loop (cost=10000000246.47..10000000277.74 rows=194 width=10) (actual time=4.775..4.887 rows=98 loops=1)
2	-> HashAggregate (cost=10000000246.47..10000000248.41 rows=194 width=5) (actual time=4.764..4.775 rows=98 loops=1)
3	Group Key: match_details.team_id
4	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=5000 width=5) (actual time=0.027..2.829 rows=5000 loops=1)
5	Filter: (((play_stage)::text = 'a'::text) AND ((win_lose)::text = 'W'::text))
6	Rows Removed by Filter: 4998
7	-> Index Scan using country_id_hash on soccer_country (cost=0.00..0.14 rows=1 width=15) (actual time=0.001..0.001 rows=1 loops=98)
8	Index Cond: (country_id = match_details.team_id)
9	Planning Time: 0.227 ms
10	Execution Time: 4.950 ms

Figure 3.b

Query Editor | Data Output | Messages | Explain | Notifications | Query History

	QUERY PLAN text
1	Nested Loop (cost=10000000246.52..10000001038.28 rows=194 width=10) (actual time=4.891..6.002 rows=98 loops=1)
2	-> HashAggregate (cost=10000000246.47..10000000248.41 rows=194 width=5) (actual time=4.839..4.915 rows=98 loops=1)
3	Group Key: match_details.team_id
4	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=5000 width=5) (actual time=0.016..2.842 rows=5000 loops...
5	Filter: (((play_stage)::text = 'a'::text) AND ((win_lose)::text = 'W'::text))
6	Rows Removed by Filter: 4998
7	-> Bitmap Heap Scan on soccer_country (cost=0.05..4.06 rows=1 width=15) (actual time=0.003..0.003 rows=1 loops=98)
8	Recheck Cond: (country_id = match_details.team_id)
9	Heap Blocks: exact=98
10	-> Bitmap Index Scan on country_id_gin (cost=0.00..0.05 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=98)
11	Index Cond: (country_id = match_details.team_id)
12	Planning Time: 0.183 ms
13	Execution Time: 6.060 ms

Figure 3.c

d. Multiple Indices

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Before Indices:*
  - *Query Plan: Figure (i).*
  - *Estimated Cost: 20000000257.46.*
  - *Average Execution Time: 5.16ms.*
- *After Indices:*
  - *Indices Used: Hash-based index on soccer_country.country_id and bitmap index on match_details.win_lose.*

- *Query Plan: Figure 4.*
- *Estimated Cost: 253.52.*
- *Average Execution Time: 3.585ms.*
- *Explanation: Since applying hash-based index on soccer_country.country_id reduced the time complexity to O (1) and applying bitmap index on match_details.win_lose also dropped the complexity due to the bit-wise that is faster.*

e. *Best case applied on the query*

According to the costs and execution time of all cases applied on the columns, the best case scenario is to apply the multiple indices case on the query since it has the least execution time and cost, as it uses the hash-based index when searching for exact values, and bitmap (GIN) index with columns that is included in AND operation.

Query Editor | Data Output | Messages | Explain | Notifications | Query History

	QUERY PLAN text
1	Nested Loop (cost=222.25..253.52 rows=194 width=10) (actual time=4.297..4.447 rows=98 loops=1)
2	-> HashAggregate (cost=222.25..224.19 rows=194 width=5) (actual time=4.287..4.301 rows=98 loops=1)
3	Group Key: match_details.team_id
4	-> Bitmap Heap Scan on match_details (cost=50.75..209.75 rows=5000 width=5) (actual time=1.077..2.233 rows=5000 loops=1)
5	Recheck Cond: ((win_lose)::text = 'W'::text)
6	Filter: ((play_stage)::text = 'a'::text)
7	Heap Blocks: exact=84
8	-> Bitmap Index Scan on win_lose_gin (cost=0.00..49.50 rows=5000 width=0) (actual time=1.048..1.048 rows=5000 loops=1)
9	Index Cond: ((win_lose)::text = 'W'::text)
10	-> Index Scan using country_id_hash on soccer_country (cost=0.00..0.14 rows=1 width=15) (actual time=0.001..0.001 rows=1 loops=98)
11	Index Cond: (country_id = match_details.team_id)
12	Planning Time: 0.371 ms
13	Execution Time: 4.529 ms

Figure 4

3. *Query 14*

a. *Index on match_details.team_id*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (iii).*
  - *Estimated Cost: 248.34.*
  - *Average Execution Time: 4.039ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 5.a*
  - *Estimated Cost: 113.72.*
  - *Average Execution Time: 0.28ms.*
  - *Explanation: The reason of the difference before and after applying the index is that the B-Tree has a time complexity of O (log n), and the column has no duplicates which makes the operation faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 5.b*
  - *Estimated Cost: 113.15.*
  - *Average Execution Time: 0.206ms.*
  - *Explanation: The hash-based index has time complexity of O (1), so it gives a better performance than not applying any indices, especially with having no duplicates on the column.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 5.c.*
  - *Estimated Cost: 121.14.*
  - *Average Execution Time: 0.276ms*

- *Explanation: The difference is that applying the bitmap index makes the operation faster since the bit-based operation makes the operation faster.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, but it is faster due to bit-wise operation and having some duplicates in this column, but the hash-based index is still much faster.

	QUERY PLAN text
1	GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=6.201..6.207 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.020..0.053 rows=1 loops=1)
7	Filter: ((country_name)::text = 'Germany1'::text)
8	Rows Removed by Filter: 194
9	InitPlan 2 (returns $1)
10	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.165 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany2'::text)
12	Rows Removed by Filter: 194
13	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=6.182..6.182 rows=4 loops=1)
14	Sort Key: match_details.match_no
15	Sort Method: quicksort Memory: 25kB
16	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.105..6.167 rows=4 loops=1)
17	Filter: ((team_id = $0) OR (team_id = $1))
18	Rows Removed by Filter: 9994
19	Planning Time: 2.767 ms
20	Execution Time: 6.263 ms

Figure (iii)

	QUERY PLAN text
1	GroupAggregate (cost=111.67..113.72 rows=1 width=5) (actual time=0.180..0.187 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.022..0.039 rows=1 loops=1)
7	Filter: ((country_name)::text = 'Germany1'::text)
8	Rows Removed by Filter: 194
9	InitPlan 2 (returns $1)
10	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.038 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany2'::text)
12	Rows Removed by Filter: 194
13	-> Sort (cost=102.80..103.06 rows=103 width=10) (actual time=0.166..0.167 rows=4 loops=1)
14	Sort Key: match_details.match_no
15	Sort Method: quicksort Memory: 25kB
16	-> Bitmap Heap Scan on match_details (cost=9.40..99.36 rows=103 width=10) (actual time=0.151..0.152 rows=4 loops=1)
17	Recheck Cond: ((team_id = $0) OR (team_id = $1))
18	Heap Blocks: exact=1
19	-> BitmapOr (cost=9.40..9.40 rows=103 width=0) (actual time=0.148..0.148 rows=0 loops=1)
20	-> Bitmap Index Scan on team_id_btree (cost=0.00..4.67 rows=52 width=0) (actual time=0.103..0.103 rows=2 loops=1)
21	Index Cond: (team_id = $0)
22	-> Bitmap Index Scan on team_id_btree (cost=0.00..4.67 rows=52 width=0) (actual time=0.043..0.043 rows=2 loops=1)
23	Index Cond: (team_id = $1)
24	Planning Time: 1.692 ms
25	Execution Time: 0.272 ms

Figure 5.a

	QUERY PLAN text
1	GroupAggregate (cost=111.10..113.15 rows=1 width=5) (actual time=0.107..0.112 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.037 rows=1 loops=1)
7	Filter: ((country_name)::text = 'Germany1'::text)
8	Rows Removed by Filter: 194
9	InitPlan 2 (returns $1)
10	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany2'::text)
12	Rows Removed by Filter: 194
13	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.098..0.098 rows=4 loops=1)
14	Sort Key: match_details.match_no
15	Sort Method: quicksort Memory: 25kB
16	-> Bitmap Heap Scan on match_details (cost=8.83..98.79 rows=103 width=10) (actual time=0.088..0.090 rows=4 loops=1)
17	Recheck Cond: ((team_id = $0) OR (team_id = $1))
18	Heap Blocks: exact=1
19	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.084..0.084 rows=0 loops=1)
20	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.054..0.054 rows=2 loops=1)
21	Index Cond: (team_id = $0)
22	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.030..0.030 rows=2 loops=1)
23	Index Cond: (team_id = $1)
24	Planning Time: 1.536 ms
25	Execution Time: 0.180 ms

Figure 5.b

	QUERY PLAN text
1	GroupAggregate (cost=119.10..121.14 rows=1 width=5) (actual time=0.102..0.106 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.034 rows=1 loops=1)
7	Filter: ((country_name)::text = 'Germany1'::text)
8	Rows Removed by Filter: 194
9	InitPlan 2 (returns $1)
10	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany2'::text)
12	Rows Removed by Filter: 194
13	-> Sort (cost=110.22..110.48 rows=103 width=10) (actual time=0.092..0.093 rows=4 loops=1)
14	Sort Key: match_details.match_no
15	Sort Method: quicksort Memory: 25kB
16	-> Bitmap Heap Scan on match_details (cost=16.82..106.78 rows=103 width=10) (actual time=0.082..0.082 rows=4 loops=1)
17	Recheck Cond: ((team_id = $0) OR (team_id = $1))
18	Heap Blocks: exact=1
19	-> BitmapOr (cost=16.82..16.82 rows=103 width=0) (actual time=0.079..0.079 rows=0 loops=1)
20	-> Bitmap Index Scan on team_id_gin (cost=0.00..8.39 rows=52 width=0) (actual time=0.050..0.050 rows=2 loops=1)
21	Index Cond: (team_id = $0)
22	-> Bitmap Index Scan on team_id_gin (cost=0.00..8.39 rows=52 width=0) (actual time=0.029..0.029 rows=2 loops=1)
23	Index Cond: (team_id = $1)
24	Planning Time: 1.695 ms
25	Execution Time: 0.158 ms

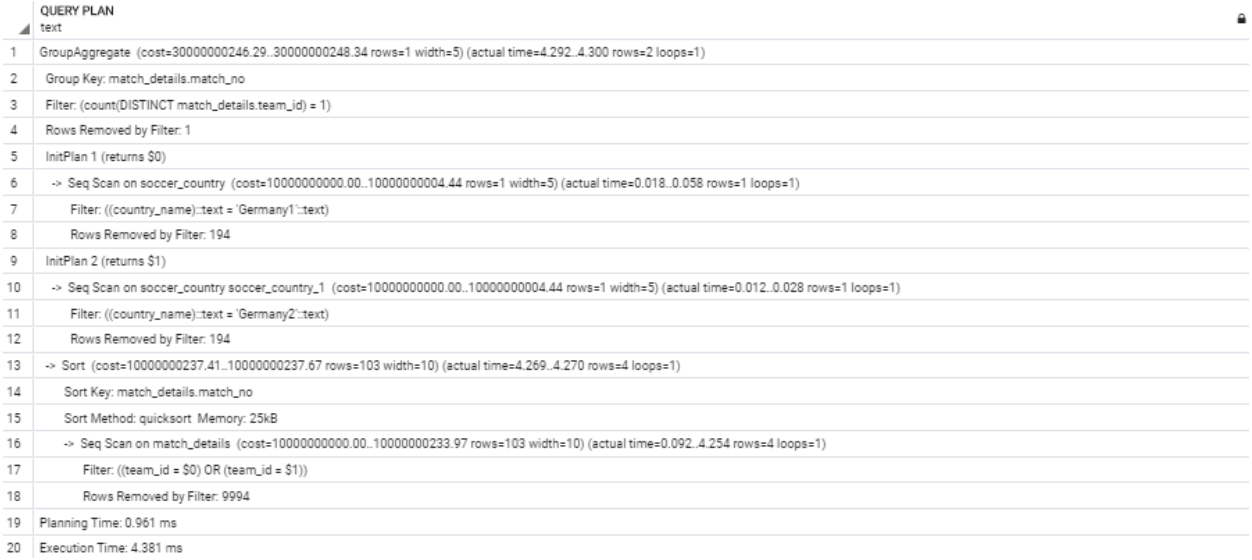
Figure 5.c

b. Index on soccer_country.country_name

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Without Indices:*
  - *Query Plan: Figure (iv).*
  - *Estimated Cost: 30000000248.34.*
  - *Average Execution Time: 4.08ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 6.a*
  - *Estimated Cost: 10000000255.76.*
  - *Average Execution Time: 3.37ms.*
  - *Explanation: The difference is that B-Tree has a time complexity of O (log n), which is faster than the linear search before applying the index.*
- *Hash-based Index:*
  - *Query Plan: Figure 6.b*
  - *Estimated Cost: 10000000255.5.*
  - *Average Execution Time: 3.17ms.*
  - *Explanation: The difference is that the time complexity is O (1), which is faster than not applying the index and also the column has no duplicates while inserting the data which makes the operation much faster.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 6.c.*
  - *Estimated Cost: 10000000263.5.*
  - *Average Execution Time: 3.589ms.*
  - *Explanation: It is pretty faster than without any indices because of the fast of the bit-based operation of the bitmap index.*

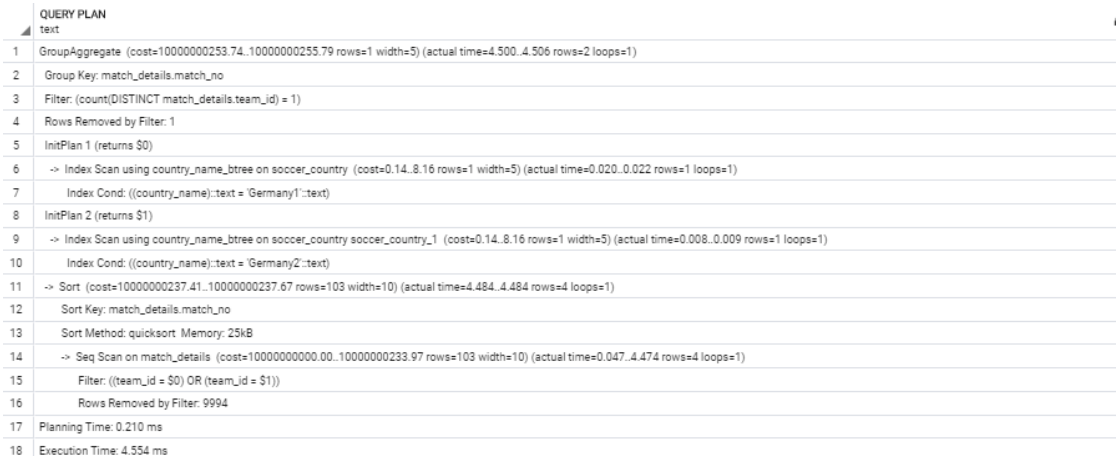
- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).



	QUERY PLAN text
1	GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=4.292..4.300 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.018..0.058 rows=1 loops=1)
7	Filter: ((country_name)::text = 'Germany1'::text)
8	Rows Removed by Filter: 194
9	InitPlan 2 (returns $1)
10	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany2'::text)
12	Rows Removed by Filter: 194
13	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.269..4.270 rows=4 loops=1)
14	Sort Key: match_details.match_no
15	Sort Method: quicksort Memory: 25kB
16	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.092..4.254 rows=4 loops=1)
17	Filter: ((team_id = $0) OR (team_id = $1))
18	Rows Removed by Filter: 9994
19	Planning Time: 0.961 ms
20	Execution Time: 4.381 ms

Figure (iv)



	QUERY PLAN text
1	GroupAggregate (cost=10000000253.74..10000000255.79 rows=1 width=5) (actual time=4.500..4.506 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Index Scan using country_name_btree on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.020..0.022 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1'::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using country_name_btree on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.008..0.009 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2'::text)
11	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.484..4.484 rows=4 loops=1)
12	Sort Key: match_details.match_no
13	Sort Method: quicksort Memory: 25kB
14	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.047..4.474 rows=4 loops=1)
15	Filter: ((team_id = $0) OR (team_id = $1))
16	Rows Removed by Filter: 9994
17	Planning Time: 0.210 ms
18	Execution Time: 4.554 ms

Figure 6.a

	QUERY PLAN text
1	GroupAggregate (cost=10000000253.45..10000000255.50 rows=1 width=5) (actual time=2.705..2.710 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Index Scan using country_name_hash on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.012..0.013 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1'::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using country_name_hash on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.002..0.003 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2'::text)
11	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.695..2.695 rows=4 loops=1)
12	Sort Key: match_details.match_no
13	Sort Method: quicksort Memory: 25kB
14	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.030..2.685 rows=4 loops=1)
15	Filter: ((team_id = $0) OR (team_id = $1))
16	Rows Removed by Filter: 9994
17	Planning Time: 1.399 ms
18	Execution Time: 2.744 ms

Figure 6.b

	QUERY PLAN text
1	GroupAggregate (cost=10000000261.45..10000000263.50 rows=1 width=5) (actual time=4.330..4.335 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=5) (actual time=0.034..0.036 rows=1 loops=1)
7	Recheck Cond: ((country_name)::text = 'Germany1'::text)
8	Heap Blocks: exact=1
9	-> Bitmap Index Scan on country_name_gin (cost=0.00..8.01 rows=1 width=0) (actual time=0.029..0.029 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany1'::text)
11	InitPlan 2 (returns $1)
12	-> Bitmap Heap Scan on soccer_country soccer_country_1 (cost=8.01..12.02 rows=1 width=5) (actual time=0.016..0.016 rows=1 loops=1)
13	Recheck Cond: ((country_name)::text = 'Germany2'::text)
14	Heap Blocks: exact=1
15	-> Bitmap Index Scan on country_name_gin (cost=0.00..8.01 rows=1 width=0) (actual time=0.013..0.013 rows=1 loops=1)
16	Index Cond: ((country_name)::text = 'Germany2'::text)
17	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.311..4.312 rows=4 loops=1)
18	Sort Key: match_details.match_no
19	Sort Method: quicksort Memory: 25kB
20	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.108..4.297 rows=4 loops=1)
21	Filter: ((team_id = $0) OR (team_id = $1))
22	Rows Removed by Filter: 9994
23	Planning Time: 1.629 ms
24	Execution Time: 4.429 ms

Figure 6.c

c. Multiple Indices

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Before Indices:*
  - *Query Plan: Figure (iv).*
  - *Estimated Cost: 30000000248.34.*
  - *Average Execution Time: 4.08ms.*
- *After Indices:*

- *Indices Used: Hash-based on match_details.team_id and soccer_country.country_name.*
- *Query Plan: Figure 7.*
- *Estimated Cost: 120.31.*
- *Average Execution Time: 0.1576ms.*
- *Explanation: The difference is that exchanging the linear search with the constant time complexity of the hash-based index has made the operation much faster.*

d. *Best case applied on the query*

According to the costs and execution time of all cases applied on the columns, the best-case scenario is to apply the multiple indices case on the query since it has the least execution time and cost, as it uses the hash-based index when searching for exact values.

	QUERY PLAN text
1	GroupAggregate (cost=118.26..120.31 rows=1 width=5) (actual time=0.121..0.127 rows=2 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	Rows Removed by Filter: 1
5	InitPlan 1 (returns $0)
6	-> Index Scan using country_name_hash on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.029..0.030 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1'::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using country_name_hash on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.004..0.004 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2'::text)
11	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.075..0.076 rows=4 loops=1)
12	Sort Key: match_details.match_no
13	Sort Method: quicksort Memory: 25kB
14	-> Bitmap Heap Scan on match_details (cost=8.83..98.79 rows=103 width=10) (actual time=0.062..0.064 rows=4 loops=1)
15	Recheck Cond: ((team_id = $0) OR (team_id = $1))
16	Heap Blocks: exact=1
17	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.053..0.053 rows=0 loops=1)
18	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.043..0.043 rows=2 loops=1)
19	Index Cond: (team_id = $0)
20	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.009..0.009 rows=2 loops=1)
21	Index Cond: (team_id = $1)
22	Planning Time: 0.296 ms
23	Execution Time: 0.203 ms

Figure 7

4. *Query 15*

a. *Index on match_details.team_id*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (vi).*
  - *Estimated Cost: 363.49.*
  - *Average Execution Time: 5.05ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 8.a.*
  - *Estimated Cost: 228.88.*
  - *Average Execution Time: 1.81ms.*
  - *Explanation: The reason of this decrease in both cost and execution time is that the time complexity of B-tree (O (log n)) is much less than the linear time (O (n)).*
- *Hash-based Index:*
  - *Query Plan: Figure 8.b.*
  - *Estimated Cost: 228.31.*
  - *Average Execution Time: 1.68ms.*
  - *Explanation: Due to the constant time complexity of the hash-based index compared to the linear time of the query without applying the index.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 8.c.*
  - *Estimated Cost: 236.3.*
  - *Average Execution Time: 2.33ms.*

- *Explanation: The difference is that the bitmap index stores the rows that has a specific value in a column as a vector of bits, which makes finding the rows that has the value faster than without applying the index.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, but it is faster due to bit-wise operation and having some duplicates in this column, but the hash-based index is still much faster.

	QUERY PLAN text
1	Hash Join (cost=248.36..363.49 rows=1 width=19) (actual time=2.777..3.719 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=19) (actual time=0.018..0.350 rows=5000 loops=1)
4	-> Hash (cost=248.35..248.35 rows=1 width=5) (actual time=2.753..2.753 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=2.747..2.751 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.031 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany1'::text)
13	Rows Removed by Filter: 194
14	InitPlan 2 (returns $1)
15	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany2'::text)
17	Rows Removed by Filter: 194
18	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=2.737..2.737 rows=4 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 25kB
21	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.048..2.729 rows=4 loops=1)
22	Filter: ((team_id = $0) OR (team_id = $1))
23	Rows Removed by Filter: 9994
24	Planning Time: 0.221 ms
25	Execution Time: 3.759 ms

Figure (vi)

	QUERY PLAN text
1	Hash Join (cost=113.74..228.88 rows=1 width=19) (actual time=0.298..2.143 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=19) (actual time=0.020..0.702 rows=5000 loops=1)
4	-> Hash (cost=113.73..113.73 rows=1 width=5) (actual time=0.262..0.263 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=111.67..113.72 rows=1 width=5) (actual time=0.232..0.246 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.021..0.065 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany1'::text)
13	Rows Removed by Filter: 194
14	InitPlan 2 (returns $1)
15	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.017..0.058 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany2'::text)
17	Rows Removed by Filter: 194
18	-> Sort (cost=102.80..103.06 rows=103 width=10) (actual time=0.211..0.212 rows=4 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 25kB
21	-> Bitmap Heap Scan on match_details (cost=9.40..99.36 rows=103 width=10) (actual time=0.193..0.197 rows=4 loops=1)
22	Recheck Cond: ((team_id = $0) OR (team_id = $1))
23	Heap Blocks: exact=1
24	-> BitmapOr (cost=9.40..9.40 rows=103 width=0) (actual time=0.190..0.190 rows=0 loops=1)
25	-> Bitmap Index Scan on team_id_btree (cost=0.00..4.67 rows=52 width=0) (actual time=0.125..0.125 rows=2 loops=1)
26	Index Cond: (team_id = $0)
27	-> Bitmap Index Scan on team_id_btree (cost=0.00..4.67 rows=52 width=0) (actual time=0.064..0.064 rows=2 loops=1)
28	Index Cond: (team_id = $1)
29	Planning Time: 2.379 ms
30	Execution Time: 2.289 ms

Figure 8.a

	QUERY PLAN text
1	Hash Join (cost=113.17..228.31 rows=1 width=19) (actual time=0.178..1.797 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=19) (actual time=0.019..0.620 rows=5000 loops=1)
4	-> Hash (cost=113.16..113.16 rows=1 width=5) (actual time=0.149..0.149 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=111.10..113.15 rows=1 width=5) (actual time=0.139..0.146 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.020..0.046 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany1'::text)
13	Rows Removed by Filter: 194
14	InitPlan 2 (returns $1)
15	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.037 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany2'::text)
17	Rows Removed by Filter: 194
18	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.126..0.127 rows=4 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 25kB
21	-> Bitmap Heap Scan on match_details (cost=8.83..98.79 rows=103 width=10) (actual time=0.115..0.117 rows=4 loops=1)
22	Recheck Cond: ((team_id = $0) OR (team_id = $1))
23	Heap Blocks: exact=1
24	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.110..0.110 rows=0 loops=1)
25	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.066..0.066 rows=2 loops=1)
26	Index Cond: (team_id = $0)
27	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.043..0.044 rows=2 loops=1)
28	Index Cond: (team_id = $1)
29	Planning Time: 2.289 ms
30	Execution Time: 1.909 ms

Figure 8.b

	QUERY PLAN text
1	Hash Join (cost=121.17..236.30 rows=1 width=19) (actual time=0.127..1.351 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=19) (actual time=0.012..0.486 rows=5000 loops=1)
4	-> Hash (cost=121.15..121.15 rows=1 width=5) (actual time=0.102..0.102 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=119.10..121.14 rows=1 width=5) (actual time=0.094..0.099 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.031 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany1'::text)
13	Rows Removed by Filter: 194
14	InitPlan 2 (returns $1)
15	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany2'::text)
17	Rows Removed by Filter: 194
18	-> Sort (cost=110.22..110.48 rows=103 width=10) (actual time=0.085..0.085 rows=4 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 25kB
21	-> Bitmap Heap Scan on match_details (cost=16.82..106.78 rows=103 width=10) (actual time=0.076..0.077 rows=4 loops=1)
22	Recheck Cond: ((team_id = $0) OR (team_id = $1))
23	Heap Blocks: exact=1
24	-> BitmapOr (cost=16.82..16.82 rows=103 width=0) (actual time=0.074..0.074 rows=0 loops=1)
25	-> Bitmap Index Scan on team_id_gin (cost=0.00..8.39 rows=52 width=0) (actual time=0.044..0.044 rows=2 loops=1)
26	Index Cond: (team_id = $0)
27	-> Bitmap Index Scan on team_id_gin (cost=0.00..8.39 rows=52 width=0) (actual time=0.029..0.029 rows=2 loops=1)
28	Index Cond: (team_id = $1)
29	Planning Time: 0.226 ms
30	Execution Time: 1.446 ms

Figure 8.c

b. Index on soccer_country.country_name

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Without Indices:*
  - *Query Plan: Figure (vii).*
  - *Estimated Cost: 40000000363.49.*
  - *Average Execution Time: 8.12ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 9.a*
  - *Estimated Cost: 20000000370.94.*
  - *Average Execution Time: 5.4982ms.*
  - *Explanation: The reason of this difference is that B-Tree has a faster time complexity of O (log n) and there are no duplicates in the insertions of this column.*

- *Hash-based Index:*
  - *Query Plan: Figure 9.b*
  - *Estimated Cost: 20000000370.65.*
  - *Average Execution Time: 4.7594ms.*
  - *Explanation: Due to the constant time complexity of the hash-based index compared to the linear time of the query without any indices.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 9.c.*
  - *Estimated Cost: 20000000378.66.*
  - *Average Execution Time: 7.124ms.*
  - *Explanation: The reason behind this difference is that the bitmap index has a faster time due to its bit-wise operation, but it is not too much fast since this column has no duplicates in it.*
- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).

	QUERY PLAN text
1	Hash Join (cost=40000000248.36..40000000363.49 rows=1 width=19) (actual time=5.442..7.446 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=19) (actual time=0.025..0.737 rows=5000 loops=1)
4	-> Hash (cost=30000000248.35..30000000248.35 rows=1 width=5) (actual time=5.369..5.370 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=5.353..5.363 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.017..0.037 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany1'::text)
13	Rows Removed by Filter: 194
14	InitPlan 2 (returns $1)
15	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany2'::text)
17	Rows Removed by Filter: 194
18	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=5.323..5.324 rows=4 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 25kB
21	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.063..5.306 rows=4 loops=1)
22	Filter: ((team_id = $0) OR (team_id = $1))
23	Rows Removed by Filter: 9994
24	Planning Time: 0.277 ms
25	Execution Time: 7.513 ms

Figure (vii)

	QUERY PLAN text
1	Hash Join (cost=20000000255.81..20000000370.94 rows=1 width=19) (actual time=5.585..7.245 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=19) (actual time=0.028..0.661 rows=5000 loops=1)
4	-> Hash (cost=10000000255.80..10000000255.80 rows=1 width=5) (actual time=5.544..5.544 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=10000000253.74..10000000255.79 rows=1 width=5) (actual time=5.529..5.538 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Index Scan using country_name_btree on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.118..0.120 rows=1 loops=1)
12	Index Cond: ((country_name)::text = 'Germany1'::text)
13	InitPlan 2 (returns $1)
14	-> Index Scan using country_name_btree on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.045..0.046 rows=1 loops=1)
15	Index Cond: ((country_name)::text = 'Germany2'::text)
16	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=5.497..5.498 rows=4 loops=1)
17	Sort Key: match_details.match_no
18	Sort Method: quicksort Memory: 25kB
19	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.159..5.481 rows=4 loops=1)
20	Filter: ((team_id = $0) OR (team_id = $1))
21	Rows Removed by Filter: 9994
22	Planning Time: 3.941 ms
23	Execution Time: 7.320 ms

Figure 9.a

	QUERY PLAN text
1	Hash Join (cost=20000000255.52..20000000370.65 rows=1 width=19) (actual time=2.786..3.771 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=19) (actual time=0.014..0.402 rows=5000 loops=1)
4	-> Hash (cost=10000000255.51..10000000255.51 rows=1 width=5) (actual time=2.765..2.766 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=10000000253.45..10000000255.50 rows=1 width=5) (actual time=2.758..2.763 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Index Scan using country_name_hash on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.017..0.017 rows=1 loops=1)
12	Index Cond: ((country_name)::text = 'Germany1'::text)
13	InitPlan 2 (returns $1)
14	-> Index Scan using country_name_hash on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
15	Index Cond: ((country_name)::text = 'Germany2'::text)
16	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.747..2.747 rows=4 loops=1)
17	Sort Key: match_details.match_no
18	Sort Method: quicksort Memory: 25kB
19	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.032..2.737 rows=4 loops=1)
20	Filter: ((team_id = $0) OR (team_id = $1))
21	Rows Removed by Filter: 9994
22	Planning Time: 1.401 ms
23	Execution Time: 3.815 ms

Figure 9.b

	QUERY PLAN text
1	Hash Join (cost=20000000263.52..20000000378.66 rows=1 width=19) (actual time=3.985..5.153 rows=2 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=19) (actual time=0.013..0.420 rows=5000 loops=1)
4	-> Hash (cost=10000000263.51..10000000263.51 rows=1 width=5) (actual time=3.963..3.963 rows=2 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> GroupAggregate (cost=10000000261.45..10000000263.50 rows=1 width=5) (actual time=3.956..3.961 rows=2 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	Rows Removed by Filter: 1
10	InitPlan 1 (returns $0)
11	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=5) (actual time=0.022..0.022 rows=1 loops=1)
12	Recheck Cond: ((country_name)::text = 'Germany1'::text)
13	Heap Blocks: exact=1
14	-> Bitmap Index Scan on country_name_gin (cost=0.00..8.01 rows=1 width=0) (actual time=0.019..0.019 rows=1 loops=1)
15	Index Cond: ((country_name)::text = 'Germany1'::text)
16	InitPlan 2 (returns $1)
17	-> Bitmap Heap Scan on soccer_country soccer_country_1 (cost=8.01..12.02 rows=1 width=5) (actual time=0.004..0.004 rows=1 loops=1)
18	Recheck Cond: ((country_name)::text = 'Germany2'::text)
19	Heap Blocks: exact=1
20	-> Bitmap Index Scan on country_name_gin (cost=0.00..8.01 rows=1 width=0) (actual time=0.004..0.004 rows=1 loops=1)
21	Index Cond: ((country_name)::text = 'Germany2'::text)
22	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.943..3.944 rows=4 loops=1)
23	Sort Key: match_details.match_no
24	Sort Method: quicksort Memory: 25kB
25	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.037..3.936 rows=4 loops=1)
26	Filter: ((team_id = $0) OR (team_id = $1))
27	Rows Removed by Filter: 9994
28	Planning Time: 1.943 ms
29	Execution Time: 5.249 ms

Figure 9.c

c. *Index on match_mast.match_no*

- *Flags Modifications: The flags enable_hashjoin and enable_mergejoin were set off.*
- *Without Indices:*
  - *Query Plan: Figure (viii).*
  - *Estimated Cost: 412.85.*
  - *Average Execution Time: 9.02ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 10.a*
  - *Estimated Cost: 256.66.*
  - *Average Execution Time: 4.67ms.*
  - *Explanation: B-tree index has a time complexity of O (log n), this makes the cost and the average execution time faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 10.b*
  - *Estimated Cost: 256.37.*
  - *Average Execution Time: 4.084ms.*
- *Explanation: Hash-based index has a time complexity of O (1) and that makes the cost and the average execution time even faster than not applying any indices on it.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 10.c.*
  - *Estimated Cost: 264.38.*
  - *Average Execution Time: 6.52ms.*

- *Explanation: Bitmap (GIN) index makes the operation faster due to the bit-wise operation of it, but it works much faster with the duplicates, which is not the case with this column.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).

	QUERY PLAN text
1	Nested Loop (cost=246.29..412.85 rows=1 width=19) (actual time=5.346..11.317 rows=2 loops=1)
2	Join Filter: (match_mast.match_no = match_details.match_no)
3	Rows Removed by Join Filter: 9998
4	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=5.307..5.335 rows=2 loops=1)
5	Group Key: match_details.match_no
6	Filter: (count(DISTINCT match_details.team_id) = 1)
7	Rows Removed by Filter: 1
8	InitPlan 1 (returns $0)
9	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.033 rows=1 loops=1)
10	Filter: ((country_name)::text = 'Germany1'::text)
11	Rows Removed by Filter: 194
12	InitPlan 2 (returns $1)
13	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
14	Filter: ((country_name)::text = 'Germany2'::text)
15	Rows Removed by Filter: 194
16	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=5.283..5.285 rows=4 loops=1)
17	Sort Key: match_details.match_no
18	Sort Method: quicksort Memory: 25kB
19	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.054..5.271 rows=4 loops=1)
20	Filter: ((team_id = $0) OR (team_id = $1))
21	Rows Removed by Filter: 9994
22	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=19) (actual time=0.020..0.845 rows=5000 loops=2)
23	Planning Time: 1.080 ms
24	Execution Time: 11.373 ms

Figure (viii)

	QUERY PLAN text
1	Nested Loop (cost=246.57..256.66 rows=1 width=19) (actual time=4.487..4.498 rows=2 loops=1)
2	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.439..4.446 rows=2 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	Rows Removed by Filter: 1
6	InitPlan 1 (returns $0)
7	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.033 rows=1 loops=1)
8	Filter: ((country_name)::text = 'Germany1'::text)
9	Rows Removed by Filter: 194
10	InitPlan 2 (returns $1)
11	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany2'::text)
13	Rows Removed by Filter: 194
14	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.412..4.413 rows=4 loops=1)
15	Sort Key: match_details.match_no
16	Sort Method: quicksort Memory: 25kB
17	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.055..4.400 rows=4 loops=1)
18	Filter: ((team_id = $0) OR (team_id = $1))
19	Rows Removed by Filter: 9994
20	-> Index Scan using match_no_btree on match_mast (cost=0.28..8.30 rows=1 width=19) (actual time=0.024..0.024 rows=1 loops=2)
21	Index Cond: (match_no = match_details.match_no)
22	Planning Time: 1.802 ms
23	Execution Time: 4.551 ms

Figure 10.a

	QUERY PLAN text
1	Nested Loop (cost=246.29..256.37 rows=1 width=19) (actual time=6.837..6.869 rows=2 loops=1)
2	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=6.791..6.810 rows=2 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	Rows Removed by Filter: 1
6	InitPlan 1 (returns $0)
7	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.017..0.063 rows=1 loops=1)
8	Filter: ((country_name)::text = 'Germany1'::text)
9	Rows Removed by Filter: 194
10	InitPlan 2 (returns $1)
11	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.057 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany2'::text)
13	Rows Removed by Filter: 194
14	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=6.744..6.745 rows=4 loops=1)
15	Sort Key: match_details.match_no
16	Sort Method: quicksort Memory: 25kB
17	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.090..6.727 rows=4 loops=1)
18	Filter: ((team_id = $0) OR (team_id = $1))
19	Rows Removed by Filter: 9994
20	-> Index Scan using match_no_hash on match_mast (cost=0.00..8.02 rows=1 width=19) (actual time=0.022..0.023 rows=1 loops=2)
21	Index Cond: (match_no = match_details.match_no)
22	Planning Time: 2.166 ms
23	Execution Time: 6.938 ms

Figure 10.b

	QUERY PLAN text
1	Nested Loop (cost=258.30..264.38 rows=1 width=19) (actual time=7.489..7.528 rows=2 loops=1)
2	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=7.402..7.417 rows=2 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	Rows Removed by Filter: 1
6	InitPlan 1 (returns $0)
7	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.025..0.089 rows=1 loops=1)
8	Filter: ((country_name)::text = 'Germany1'::text)
9	Rows Removed by Filter: 194
10	InitPlan 2 (returns $1)
11	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.057 rows=1 loops=1)
12	Filter: ((country_name)::text = 'Germany2'::text)
13	Rows Removed by Filter: 194
14	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=7.357..7.358 rows=4 loops=1)
15	Sort Key: match_details.match_no
16	Sort Method: quicksort Memory: 25kB
17	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.127..7.336 rows=4 loops=1)
18	Filter: ((team_id = $0) OR (team_id = $1))
19	Rows Removed by Filter: 9994
20	-> Bitmap Heap Scan on match_mast (cost=12.01..16.02 rows=1 width=19) (actual time=0.025..0.025 rows=1 loops=2)
21	Recheck Cond: (match_no = match_details.match_no)
22	Heap Blocks: exact=2
23	-> Bitmap Index Scan on match_no_gin (cost=0.00..12.01 rows=1 width=0) (actual time=0.020..0.020 rows=1 loops=2)
24	Index Cond: (match_no = match_details.match_no)
25	Planning Time: 3.992 ms

Figure 10.c

d. Multiple Indices

- *Flags Modifications: The flag enable_seqscan was set off.*
- *Before Indices:*
  - *Query Plan: Figure (vii).*
  - *Estimated Cost: 40000000363.49.*
  - *Average Execution Time: 8.12ms.*
- *After Indices:*
  - *Indices Used: Hash-based on match_details.team_id, soccer_country.country_name, and match_mast.match_no.*

- *Query Plan: Figure 11.*
- *Estimated Cost: 128.35.*
- *Average Execution Time: 0.183ms.*
- *Explanation: Due to the constant time complexity of the hash-based index that applied on the three columns.*

e. *Best case applied on the query*

According to the costs and execution time of all cases applied on the columns, the best-case scenario is to apply the multiple indices case on the query since it has the least execution time and cost, as it uses the hash-based index when searching for exact values.

	QUERY PLAN text
1	Nested Loop (cost=118.26..128.35 rows=1 width=19) (actual time=0.085..0.111 rows=2 loops=1)
2	-> GroupAggregate (cost=118.26..120.31 rows=1 width=5) (actual time=0.076..0.094 rows=2 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	Rows Removed by Filter: 1
6	InitPlan 1 (returns $0)
7	-> Index Scan using country_name_hash on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.020..0.021 rows=1 loops=1)
8	Index Cond: ((country_name)::text = 'Germany1'::text)
9	InitPlan 2 (returns $1)
10	-> Index Scan using country_name_hash on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.005..0.005 rows=1 loops=1)
11	Index Cond: ((country_name)::text = 'Germany2'::text)
12	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.053..0.054 rows=4 loops=1)
13	Sort Key: match_details.match_no
14	Sort Method: quicksort Memory: 25kB
15	-> Bitmap Heap Scan on match_details (cost=8.83..98.79 rows=103 width=10) (actual time=0.041..0.043 rows=4 loops=1)
16	Recheck Cond: ((team_id = $0) OR (team_id = $1))
17	Heap Blocks: exact=1
18	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.037..0.037 rows=0 loops=1)
19	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.029..0.029 rows=2 loops=1)
20	Index Cond: (team_id = $0)
21	-> Bitmap Index Scan on team_id_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.008..0.008 rows=2 loops=1)
22	Index Cond: (team_id = $1)
23	-> Index Scan using match_no_hash on match_mast (cost=0.00..8.02 rows=1 width=19) (actual time=0.005..0.006 rows=1 loops=2)
24	Index Cond: (match_no = match_details.match_no)
25	Planning Time: 0.283 ms
26	Execution Time: 0.173 ms

Figure 11

5. *Query 16*

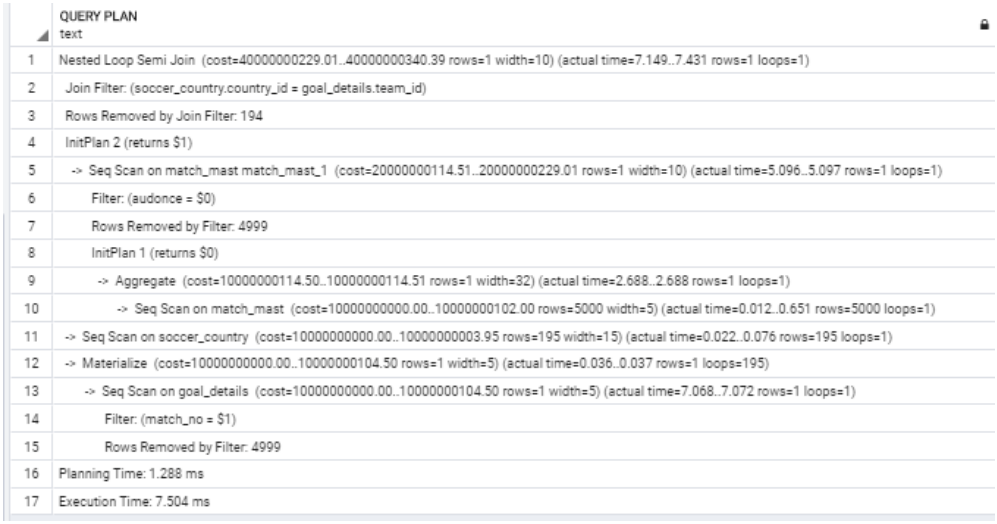
a. *Index on soccer_country.country_id*

- *Flags Modifications: The flags enable_seqscan, enable_hashjoin, and enable_mergejoin were set off.*
- *Without Indices:*
  - *Query Plan: Figure (ix).*
  - *Estimated Cost: 40000000340.39.*
  - *Average Execution Time: 6.657ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 12.a*
  - *Estimated Cost: 30000000341.7.*
  - *Average Execution Time: 5.03ms.*
- *Explanation: Since it gives a logarithmic time complexity which makes the query finishes executing faster, and also there is no duplicates in this column in this table which makes it even faster. Hash-based Index:*
  - *Query Plan: Figure 12.b*
  - *Estimated Cost: 30000000341.55.*
  - *Average Execution Time: 4.398ms.*
  - *Explanation: Since it gives a constant time complexity which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 12.c.*
  - *Estimated Cost: 30000000345.55.*
  - *Average Execution Time: 5.699ms.*

- *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster, but it is not working that efficient with columns that has no duplicates, so it is not so much fast as applying B-tree or hash-based index on this column.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).



	QUERY PLAN text
1	Nested Loop Semi Join (cost=40000000229.01..40000000340.39 rows=1 width=10) (actual time=7.149..7.431 rows=1 loops=1)
2	Join Filter: (soccer_country.country_id = goal_details.team_id)
3	Rows Removed by Join Filter: 194
4	InitPlan 2 (returns $1)
5	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=5.096..5.097 rows=1 loops=1)
6	Filter: (audonce = $0)
7	Rows Removed by Filter: 4999
8	InitPlan 1 (returns $0)
9	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=2.688..2.688 rows=1 loops=1)
10	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.012..0.651 rows=5000 loops=1)
11	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.022..0.076 rows=195 loops=1)
12	-> Materialize (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=0.036..0.037 rows=1 loops=195)
13	-> Seq Scan on goal_details (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=7.068..7.072 rows=1 loops=1)
14	Filter: (match_no = $1)
15	Rows Removed by Filter: 4999
16	Planning Time: 1.288 ms
17	Execution Time: 7.504 ms

Figure (ix)

	QUERY PLAN text
1	Nested Loop (cost=30000000333.66..30000000341.70 rows=1 width=10) (actual time=4.418..4.420 rows=1 loops=1)
2	InitPlan 2 (returns $1)
3	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=3.051..3.052 rows=1 loops=1)
4	Filter: (audonce = $0)
5	Rows Removed by Filter: 4999
6	InitPlan 1 (returns $0)
7	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=1.670..1.671 rows=1 loops=1)
8	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.008..0.461 rows=5000 loops=1)
9	-> HashAggregate (cost=10000000104.50..10000000104.51 rows=1 width=5) (actual time=4.389..4.389 rows=1 loops=1)
10	Group Key: goal_details.team_id
11	-> Seq Scan on goal_details (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=4.384..4.384 rows=1 loops=1)
12	Filter: (match_no = $1)
13	Rows Removed by Filter: 4999
14	-> Index Scan using country_id_btree on soccer_country (cost=0.14..8.16 rows=1 width=15) (actual time=0.026..0.028 rows=1 loops=1)
15	Index Cond: (country_id = goal_details.team_id)
16	Planning Time: 0.265 ms
17	Execution Time: 4.467 ms

Figure 12.a

	QUERY PLAN text
1	Nested Loop (cost=30000000333.51..30000000341.55 rows=1 width=10) (actual time=6.279..6.281 rows=1 loops=1)
2	InitPlan 2 (returns $1)
3	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=4.802..4.803 rows=1 loops=1)
4	Filter: (audonce = $0)
5	Rows Removed by Filter: 4999
6	InitPlan 1 (returns $0)
7	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=3.308..3.308 rows=1 loops=1)
8	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.008..0.732 rows=5000 loops=1)
9	-> HashAggregate (cost=10000000104.50..10000000104.51 rows=1 width=5) (actual time=6.267..6.267 rows=1 loops=1)
10	Group Key: goal_details.team_id
11	-> Seq Scan on goal_details (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=6.261..6.261 rows=1 loops=1)
12	Filter: (match_no = $1)
13	Rows Removed by Filter: 4999
14	-> Index Scan using country_id_hash on soccer_country (cost=0.00..8.02 rows=1 width=15) (actual time=0.010..0.010 rows=1 loops=1)
15	Index Cond: (country_id = goal_details.team_id)
16	Planning Time: 0.345 ms
17	Execution Time: 6.352 ms

Figure 12.b

	QUERY PLAN text
1	Nested Loop (cost=30000000341.52..30000000345.55 rows=1 width=10) (actual time=4.326..4.327 rows=1 loops=1)
2	InitPlan 2 (returns $1)
3	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=2.459..2.459 rows=1 loops=1)
4	Filter: (audonce = $0)
5	Rows Removed by Filter: 4999
6	InitPlan 1 (returns $0)
7	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=1.467..1.468 rows=1 loops=1)
8	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.008..0.364 rows=5000 loops=1)
9	-> HashAggregate (cost=10000000104.50..10000000104.51 rows=1 width=5) (actual time=4.278..4.279 rows=1 loops=1)
10	Group Key: goal_details.team_id
11	-> Seq Scan on goal_details (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=4.270..4.271 rows=1 loops=1)
12	Filter: (match_no = $1)
13	Rows Removed by Filter: 4999
14	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=15) (actual time=0.013..0.013 rows=1 loops=1)
15	Recheck Cond: (country_id = goal_details.team_id)
16	Heap Blocks: exact=1
17	-> Bitmap Index Scan on country_id_gin (cost=0.00..8.01 rows=1 width=0) (actual time=0.008..0.008 rows=1 loops=1)
18	Index Cond: (country_id = goal_details.team_id)
19	Planning Time: 0.255 ms
20	Execution Time: 4.384 ms

Figure 12.c

b. *Index on goal_details.match_no*

- *Flags Modifications: The flags enable_seqscan was set off.*
- *Without Indices:*
  - *Query Plan: Figure (x).*
  - *Estimated Cost: 40000000338.*
  - *Average Execution Time: 5.8862ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 13.a*
  - *Estimated Cost: 30000000341.8.*
  - *Average Execution Time: 3.776ms.*
- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 13.b*
  - *Estimated Cost: 30000000341.51.*
  - *Average Execution Time: 3. 404ms.*
  - *Explanation: Since it gives a time complexity of O (1) which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 13.c.*
  - *Estimated Cost: 30000000349.52.*
  - *Average Execution Time: 4.196ms.*
  - *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, but since there is duplicates, it takes less time and has less cost, but not as much as the hash-based index.

	QUERY PLAN text
1	Hash Semi Join (cost=40000000333.52..40000000338.00 rows=1 width=10) (actual time=7.415..7.492 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=5.487..5.488 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=2.549..2.549 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.033..0.671 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.034..0.057 rows=195 loops=1)
11	-> Hash (cost=10000000104.50..10000000104.50 rows=1 width=5) (actual time=7.356..7.356 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on goal_details (cost=10000000000.00..10000000104.50 rows=1 width=5) (actual time=7.347..7.348 rows=1 loops=1)
14	Filter: (match_no = $1)
15	Rows Removed by Filter: 4999
16	Planning Time: 1.278 ms
17	Execution Time: 7.582 ms

Figure (x)

	QUERY PLAN text
1	Hash Semi Join (cost=30000000237.32..30000000241.80 rows=1 width=10) (actual time=2.944..2.979 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=2.819..2.819 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=1.826..1.826 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.011..0.561 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.017..0.026 rows=195 loops=1)
11	-> Hash (cost=8.30..8.30 rows=1 width=5) (actual time=2.881..2.881 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Index Scan using match_no_btree on goal_details (cost=0.28..8.30 rows=1 width=5) (actual time=2.875..2.876 rows=1 loops=1)
14	Index Cond: (match_no = $1)
15	Planning Time: 1.877 ms
16	Execution Time: 3.072 ms

Figure 13.a

	QUERY PLAN text
1	Hash Semi Join (cost=30000000237.04..30000000241.51 rows=1 width=10) (actual time=4.697..4.736 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=4.623..4.624 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=2.748..2.748 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.016..0.643 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.027..0.038 rows=195 loops=1)
11	-> Hash (cost=8.02..8.02 rows=1 width=5) (actual time=4.657..4.657 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Index Scan using match_no_hash on goal_details (cost=0.00..8.02 rows=1 width=5) (actual time=4.652..4.653 rows=1 loops=1)
14	Index Cond: (match_no = $1)
15	Planning Time: 2.186 ms
16	Execution Time: 4.833 ms

Figure 13.b

	QUERY PLAN text
1	Hash Semi Join (cost=30000000245.04..30000000249.52 rows=1 width=10) (actual time=2.330..2.361 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=20000000114.51..20000000229.01 rows=1 width=10) (actual time=2.288..2.288 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=10000000114.50..10000000114.51 rows=1 width=32) (actual time=1.237..1.238 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=10000000000.00..10000000102.00 rows=5000 width=5) (actual time=0.008..0.354 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.015..0.026 rows=195 loops=1)
11	-> Hash (cost=16.02..16.02 rows=1 width=5) (actual time=2.306..2.306 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Bitmap Heap Scan on goal_details (cost=12.01..16.02 rows=1 width=5) (actual time=2.303..2.303 rows=1 loops=1)
14	Recheck Cond: (match_no = $1)
15	Heap Blocks: exact=1
16	-> Bitmap Index Scan on match_no_gin (cost=0.00..12.01 rows=1 width=0) (actual time=2.300..2.300 rows=1 loops=1)
17	Index Cond: (match_no = $1)
18	Planning Time: 0.553 ms
19	Execution Time: 2.566 ms

Figure 13.c

c. *Index on match_mast.audonce*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xi).*
  - *Estimated Cost: 338.*
  - *Average Execution Time: 6.74ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 14.a*
  - *Estimated Cost: 117.62.*

- *Average Execution Time: 2.23ms.*

- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 14.b*
  - *Estimated Cost: 231.51.*
  - *Average Execution Time: 2.943ms.*
  - *Explanation: Since the hash-based index gives a constant time when it is applied on a column.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 14.c.*
  - *Estimated Cost: 239.52.*
  - *Average Execution Time: 3.071ms.*
  - *Explanation: Because of the fast bit-wise operation of the bitmap (GIN) index.*
- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the B-Tree index on this column, even though the hash-based index has a less time complexity. But for the aggregate functions, B-Tree index works much faster, as it can find the maximum number in an O (log n) time complexity, but the hash-based index has to search for it linearly. The same case is also applied to the bitmap index.

	QUERY PLAN text
1	Hash Semi Join (cost=333.52..338.00 rows=1 width=10) (actual time=7.606..7.676 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=114.51..229.01 rows=1 width=10) (actual time=5.349..5.349 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=4.132..4.132 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.014..0.953 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.021..0.049 rows=195 loops=1)
11	-> Hash (cost=104.50..104.50 rows=1 width=5) (actual time=7.511..7.511 rows=1 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on goal_details (cost=0.00..104.50 rows=1 width=5) (actual time=7.501..7.502 rows=1 loops=1)
14	Filter: (match_no = $1)
15	Rows Removed by Filter: 4999
16	Planning Time: 1.691 ms
17	Execution Time: 7.747 ms

Figure (xi)

	QUERY PLAN text
1	Hash Semi Join (cost=113.15..117.62 rows=1 width=10) (actual time=2.029..2.130 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 3 (returns $2)
4	-> Index Scan using aud_btree on match_mast match_mast_1 (cost=0.62..8.63 rows=1 width=10) (actual time=0.052..0.053 rows=1 loops=1)
5	Index Cond: (audonce = $1)
6	InitPlan 2 (returns $1)
7	-> Result (cost=0.32..0.33 rows=1 width=32) (actual time=0.042..0.042 rows=1 loops=1)
8	InitPlan 1 (returns $0)
9	-> Limit (cost=0.28..0.32 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
10	-> Index Only Scan Backward using aud_btree on match_mast (cost=0.28..206.78 rows=5000 width=5) (actual time=0.035..0.035 rows=1 loops=1)
11	Index Cond: (audonce IS NOT NULL)
12	Heap Fetches: 1
13	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.033..0.070 rows=195 loops=1)
14	-> Hash (cost=104.50..104.50 rows=1 width=5) (actual time=1.976..1.977 rows=1 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Seq Scan on goal_details (cost=0.00..104.50 rows=1 width=5) (actual time=1.958..1.959 rows=1 loops=1)
17	Filter: (match_no = $2)
18	Rows Removed by Filter: 4999
19	Planning Time: 0.372 ms
20	Execution Time: 2.205 ms

Figure 14.a

	QUERY PLAN text
1	Hash Semi Join (cost=227.04..231.51 rows=1 width=10) (actual time=4.558..4.612 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Index Scan using aud_hash on match_mast match_mast_1 (cost=114.51..122.53 rows=1 width=10) (actual time=3.206..3.207 rows=1 loops=1)
5	Index Cond: (audonce = $0)
6	InitPlan 1 (returns $0)
7	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=3.176..3.177 rows=1 loops=1)
8	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.014..0.754 rows=5000 loops=1)
9	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.021..0.034 rows=195 loops=1)
10	-> Hash (cost=104.50..104.50 rows=1 width=5) (actual time=4.522..4.522 rows=1 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 9kB
12	-> Seq Scan on goal_details (cost=0.00..104.50 rows=1 width=5) (actual time=4.518..4.518 rows=1 loops=1)
13	Filter: (match_no = $1)
14	Rows Removed by Filter: 4999
15	Planning Time: 1.947 ms
16	Execution Time: 4.666 ms

Figure 14.b

	QUERY PLAN text
1	Hash Semi Join (cost=235.04..239.52 rows=1 width=10) (actual time=3.003..3.044 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Bitmap Heap Scan on match_mast match_mast_1 (cost=126.52..130.53 rows=1 width=10) (actual time=1.698..1.698 rows=1 loops=1)
5	Recheck Cond: (audonce = $0)
6	Heap Blocks: exact=1
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=1.685..1.685 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.009..0.382 rows=5000 loops=1)
10	-> Bitmap Index Scan on aud_gin (cost=0.00..12.01 rows=1 width=0) (actual time=1.696..1.696 rows=1 loops=1)
11	Index Cond: (audonce = $0)
12	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.012..0.024 rows=195 loops=1)
13	-> Hash (cost=104.50..104.50 rows=1 width=5) (actual time=2.983..2.983 rows=1 loops=1)
14	Buckets: 1024 Batches: 1 Memory Usage: 9kB
15	-> Seq Scan on goal_details (cost=0.00..104.50 rows=1 width=5) (actual time=2.980..2.980 rows=1 loops=1)
16	Filter: (match_no = $1)
17	Rows Removed by Filter: 4999
18	Planning Time: 0.296 ms
19	Execution Time: 3.089 ms

Figure 14.c

d. Multiple Indices

- *Flags Modifications: All flags were set on.*
- *Before Indices:*
  - *Query Plan: Figure (xi).*
  - *Estimated Cost: 338.*
  - *Average Execution Time: 6.74ms.*
- *After Indices:*
  - *Indices Used: B-Tree index on match_mast.audonce, and hash-based index on goal_details.match_no.*
  - *Query Plan: Figure 15.*
  - *Estimated Cost: 21.14.*
  - *Average Execution Time: 0.1408ms.*
  - *Explanation: Due to the mixed usage of the B-Tree with an aggregate function and the hash-based index of a constant time complexity.*

e. Best case applied on the query

According to the costs and execution time of all cases applied on the columns, the best-case scenario is to apply the multiple indices case on the query since it has the least execution time and cost, as it uses the hash-based index when searching for exact values, and B-Tree when using aggregate functions.

	QUERY PLAN text
1	Hash Semi Join (cost=16.66..21.14 rows=1 width=10) (actual time=0.060..0.092 rows=1 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 3 (returns $2)
4	-> Index Scan using aud_btree on match_mast match_mast_1 (cost=0.62..8.63 rows=1 width=10) (actual time=0.021..0.022 rows=1 loops=1)
5	Index Cond: (audonce = $1)
6	InitPlan 2 (returns $1)
7	-> Result (cost=0.32..0.33 rows=1 width=32) (actual time=0.016..0.017 rows=1 loops=1)
8	InitPlan 1 (returns $0)
9	-> Limit (cost=0.28..0.32 rows=1 width=5) (actual time=0.014..0.014 rows=1 loops=1)
10	-> Index Only Scan Backward using aud_btree on match_mast (cost=0.28..206.78 rows=5000 width=5) (actual time=0.013..0.013 rows=1 loops=1)
11	Index Cond: (audonce IS NOT NULL)
12	Heap Fetches: 1
13	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.017..0.028 rows=195 loops=1)
14	-> Hash (cost=8.02..8.02 rows=1 width=5) (actual time=0.034..0.035 rows=1 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 9kB
16	-> Index Scan using match_no_hash on goal_details (cost=0.00..8.02 rows=1 width=5) (actual time=0.032..0.032 rows=1 loops=1)
17	Index Cond: (match_no = $2)
18	Planning Time: 1.234 ms
19	Execution Time: 0.132 ms

Figure 15

6. *Query 17*

a. *Index on player_mast.player_id*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 16.a*
  - *Estimated Cost: 771.77.*
  - *Average Execution Time: 12.363ms.*
- *Explanation: Since it gives a logarithmic time complexity which makes the query finishes executing faster, and also there is no duplicates in this column in this table which makes it even faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 16.b*
  - *Estimated Cost: 771.48.*
  - *Average Execution Time: 11.546ms.*
  - *Explanation: Since it gives a constant time complexity which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 16.c.*
  - *Estimated Cost: 779.49.*
  - *Average Execution Time: 12.772ms.*

- *Explanation: since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster, but it is not working that efficient with columns that has no duplicates, so it is not so much fast as applying B-tree or hash-based index on this column.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).

	QUERY PLAN text
1	Seq Scan on player_mast (cost=763.47..992.47 rows=1 width=9) (actual time=10.946..12.830 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=633.97..763.47 rows=1 width=5) (actual time=9.916..10.920 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.840..4.840 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.059 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.017..0.057 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.809..4.810 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.077..4.798 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.066..0.105 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.021..0.033 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.030 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $8)
37	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=4.950..4.950 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.879..3.879 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 2
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.024 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194
47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.843..3.844 rows=4 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 25kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.037..3.838 rows=4 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9994
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.073..0.115 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.026..0.040 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.038..0.038 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.033 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=4.011..4.947 rows=1 loops=1)
67	Filter: ((match_no = $6) AND (team_id = $7))
68	Rows Removed by Filter: 4999
69	Planning Time: 0.866 ms
70	Execution Time: 12.991 ms

Figure (xii)

	QUERY PLAN text
1	Index Scan using player_btree on player_mast (cost=763.75..771.77 rows=1 width=9) (actual time=11.290..11.291 rows=1 loops=1)
2	Index Cond: (player_id = $9)
3	InitPlan 10 (returns $9)
4	-> Seq Scan on goal_details goal_details_1 (cost=633.97..763.47 rows=1 width=5) (actual time=10.396..11.276 rows=1 loops=1)
5	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
6	Rows Removed by Filter: 4999
7	InitPlan 3 (returns $2)
8	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=5.278..5.278 rows=1 loops=1)
9	Group Key: match_details.match_no
10	Filter: (count(DISTINCT match_details.team_id) = 2)
11	Rows Removed by Filter: 2
12	InitPlan 1 (returns $0)
13	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.042 rows=1 loops=1)
14	Filter: ((country_name)::text = 'Germany1'::text)
15	Rows Removed by Filter: 194
16	InitPlan 2 (returns $1)
17	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.035 rows=1 loops=1)
18	Filter: ((country_name)::text = 'Germany2'::text)
19	Rows Removed by Filter: 194
20	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=5.250..5.250 rows=4 loops=1)
21	Sort Key: match_details.match_no
22	Sort Method: quicksort Memory: 25kB
23	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.072..5.240 rows=4 loops=1)
24	Filter: ((team_id = $0) OR (team_id = $1))
25	Rows Removed by Filter: 9994
26	InitPlan 4 (returns $3)
27	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.074..0.112 rows=1 loops=1)
28	Hash Cond: (b.team_id = a.country_id)
29	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.025..0.038 rows=195 loops=1)
30	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.041..0.041 rows=1 loops=1)
31	Buckets: 1024 Batches: 1 Memory Usage: 9kB
32	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.036 rows=1 loops=1)
33	Filter: ((country_name)::text = 'Germany2'::text)
34	Rows Removed by Filter: 194
35	InitPlan 9 (returns $8)
36	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=4.977..4.977 rows=1 loops=1)
37	InitPlan 7 (returns $6)
38	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.996..3.996 rows=1 loops=1)
39	Group Key: match_details_1.match_no
40	Filter: (count(DISTINCT match_details_1.team_id) = 2)
41	Rows Removed by Filter: 2
42	InitPlan 5 (returns $4)
43	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
44	Filter: ((country_name)::text = 'Germany1'::text)
45	Rows Removed by Filter: 194
46	InitPlan 6 (returns $5)
47	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.027 rows=1 loops=1)
48	Filter: ((country_name)::text = 'Germany2'::text)
49	Rows Removed by Filter: 194
50	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.966..3.966 rows=4 loops=1)
51	Sort Key: match_details_1.match_no
52	Sort Method: quicksort Memory: 25kB
53	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.039..3.960 rows=4 loops=1)
54	Filter: ((team_id = $4) OR (team_id = $5))
55	Rows Removed by Filter: 9994
56	InitPlan 8 (returns $7)
57	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.069..0.103 rows=1 loops=1)
58	Hash Cond: (b_1.team_id = a_1.country_id)
59	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.024..0.035 rows=195 loops=1)
60	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
61	Buckets: 1024 Batches: 1 Memory Usage: 9kB
62	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.030 rows=1 loops=1)
63	Filter: ((country_name)::text = 'Germany2'::text)
64	Rows Removed by Filter: 194
65	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=4.116..4.974 rows=1 loops=1)
66	Filter: ((match_no = $6) AND (team_id = $7))
67	Rows Removed by Filter: 4999
68	Planning Time: 1.115 ms
69	Execution Time: 11.439 ms

Figure 16.a

	QUERY PLAN text
1	Index Scan using player_hash on player_mast (cost=763.47..771.48 rows=1 width=9) (actual time=10.725..10.725 rows=1 loops=1)
2	Index Cond: (player_id = $9)
3	InitPlan 10 (returns $9)
4	-> Seq Scan on goal_details goal_details_1 (cost=633.97..763.47 rows=1 width=5) (actual time=9.439..10.696 rows=1 loops=1)
5	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
6	Rows Removed by Filter: 4999
7	InitPlan 3 (returns $2)
8	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.590..3.590 rows=1 loops=1)
9	Group Key: match_details.match_no
10	Filter: (count(DISTINCT match_details.team_id) = 2)
11	Rows Removed by Filter: 2
12	InitPlan 1 (returns $0)
13	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.017..0.035 rows=1 loops=1)
14	Filter: ((country_name)::text = 'Germany1'::text)
15	Rows Removed by Filter: 194
16	InitPlan 2 (returns $1)
17	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
18	Filter: ((country_name)::text = 'Germany2'::text)
19	Rows Removed by Filter: 194
20	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.567..3.567 rows=4 loops=1)
21	Sort Key: match_details.match_no
22	Sort Method: quicksort Memory: 25kB
23	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.052..3.556 rows=4 loops=1)
24	Filter: ((team_id = $0) OR (team_id = $1))
25	Rows Removed by Filter: 9994
26	InitPlan 4 (returns $3)
27	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.195..0.244 rows=1 loops=1)
28	Hash Cond: (b.team_id = a.country_id)
29	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.018..0.034 rows=195 loops=1)
30	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.167..0.167 rows=1 loops=1)
31	Buckets: 1024 Batches: 1 Memory Usage: 9kB
32	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.022..0.161 rows=1 loops=1)
33	Filter: ((country_name)::text = 'Germany2'::text)
34	Rows Removed by Filter: 194
35	InitPlan 9 (returns $8)
36	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=5.579..5.579 rows=1 loops=1)
37	InitPlan 7 (returns $6)
38	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.260..4.260 rows=1 loops=1)
39	Group Key: match_details_1.match_no
40	Filter: (count(DISTINCT match_details_1.team_id) = 2)
41	Rows Removed by Filter: 2
42	InitPlan 5 (returns $4)
43	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
44	Filter: ((country_name)::text = 'Germany1'::text)
45	Rows Removed by Filter: 194
46	InitPlan 6 (returns $5)
47	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
48	Filter: ((country_name)::text = 'Germany2'::text)
49	Rows Removed by Filter: 194
50	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.230..4.231 rows=4 loops=1)
50	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.230..4.231 rows=4 loops=1)
51	Sort Key: match_details_1.match_no
52	Sort Method: quicksort Memory: 25kB
53	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.039..4.224 rows=4 loops=1)
54	Filter: ((team_id = $4) OR (team_id = $5))
55	Rows Removed by Filter: 9994
56	InitPlan 8 (returns $7)
57	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.070..0.108 rows=1 loops=1)
58	Hash Cond: (b_1.team_id = a_1.country_id)
59	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.025..0.037 rows=195 loops=1)
60	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.038..0.038 rows=1 loops=1)
61	Buckets: 1024 Batches: 1 Memory Usage: 9kB
62	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.031 rows=1 loops=1)
63	Filter: ((country_name)::text = 'Germany2'::text)
64	Rows Removed by Filter: 194
65	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=4.384..5.575 rows=1 loops=1)
66	Filter: ((match_no = $6) AND (team_id = $7))
67	Rows Removed by Filter: 4999
68	Planning Time: 2.656 ms
69	Execution Time: 10.962 ms

Figure 16.b

	QUERY PLAN text
1	Bitmap Heap Scan on player_mast (cost=775.47..779.49 rows=1 width=9) (actual time=12.134..12.135 rows=1 loops=1)
2	Recheck Cond: (player_id = $9)
3	Heap Blocks: exact=1
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=633.97..763.47 rows=1 width=5) (actual time=10.024..12.112 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.972..4.973 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.064 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.057 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.935..4.936 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.087..4.923 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.092..0.146 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.034..0.053 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.047..0.047 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.043 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $8)
37	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=4.877..4.877 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.311..3.312 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 2
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.037 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194
47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.036 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.284..3.285 rows=4 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 25kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.060..3.277 rows=4 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9994
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.164..0.215 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.022..0.038 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.082..0.082 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.024..0.075 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=3.548..4.873 rows=1 loops=1)
67	Filter: ((match_no = $6) AND (team_id = $7))
68	Rows Removed by Filter: 4999
69	-> Bitmap Index Scan on player_gin (cost=0.00..12.01 rows=1 width=0) (actual time=12.131..12.131 rows=1 loops=1)
70	Index Cond: (player_id = $9)
71	Planning Time: 2.678 ms
72	Execution Time: 12.382 ms

Figure 16.c

b. Index on goal_details.match_no

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 17.a*
  - *Estimated Cost: 762.59.*
  - *Average Execution Time: 12.396ms.*
- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster, and also there is no duplicates in this column in this table which makes it even faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 17.b*
  - *Estimated Cost: 762.01.*
  - *Average Execution Time: 12.04ms.*
  - *Explanation: Since it gives a time complexity of O (1) which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 17.c.*
  - *Estimated Cost: 778.02.*
  - *Average Execution Time: 12.925ms.*
  - *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster, but it is*

not working that efficient with columns that has no duplicates, so it is not so much fast as applying B-tree or hash-based index on this column.

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).

	QUERY PLAN text
1	Seq Scan on player_mast (cost=533.57..762.57 rows=1 width=9) (actual time=8.807..10.919 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Index Scan using match_no_btree on goal_details goal_details_1 (cost=525.55..533.57 rows=1 width=5) (actual time=8.786..8.786 rows=1 loops=1)
6	Index Cond: (match_no = $2)
7	Filter: ((team_id = $3) AND (goal_time = $8))
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.608..3.608 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.032 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.587..3.587 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.047..3.579 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.059..0.096 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.016..0.029 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.033..0.033 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.029 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $8)
37	-> Aggregate (cost=266.78..266.79 rows=1 width=32) (actual time=5.067..5.067 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.933..4.933 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 2
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194
47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.027 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.902..4.903 rows=4 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 25kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.037..4.896 rows=4 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9994
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.072..0.110 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.025..0.038 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.035..0.035 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.031 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Index Scan using match_no_btree on goal_details (cost=0.28..8.30 rows=1 width=5) (actual time=5.063..5.064 rows=1 loops=1)
67	Index Cond: (match_no = $6)
68	Filter: (team_id = $7)
69	Planning Time: 0.542 ms
70	Execution Time: 11.125 ms

Figure 17.a

```
Seq Scan on player_mast  (cost=533.01..762.01 rows=1 width=9) (actual time=9.505..13.738 rows=1 loops=1)
 Filter: (player_id = $9)
 Rows Removed by Filter: 9999
 InitPlan 10 (returns $9)
  -> Index Scan using match_no_hash on goal_details goal_details_1  (cost=524.99..533.01 rows=1 width=5) (actual time=9.475..9.475 rows=1 loops=1)
      Index Cond: (match_no = $2)
      Filter: ((team_id = $3) AND (goal_time = $8))
      InitPlan 3 (returns $2)
       -> GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=5.374..5.374 rows=1 loops=1)
          Group Key: match_details.match_no
          Filter: (count(DISTINCT match_details.team_id) = 2)
          Rows Removed by Filter: 2
          InitPlan 1 (returns $0)
           -> Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.031..0.110 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany1'::text)
              Rows Removed by Filter: 194
          InitPlan 2 (returns $1)
           -> Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.035..0.092 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany2'::text)
              Rows Removed by Filter: 194
          -> Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=5.309..5.311 rows=4 loops=1)
              Sort Key: match_details.match_no
              Sort Method: quicksort  Memory: 25kB
              -> Seq Scan on match_details  (cost=0.00..233.97 rows=103 width=10) (actual time=0.135..5.292 rows=4 loops=1)
                 Filter: ((team_id = $0) OR (team_id = $1))
                 Rows Removed by Filter: 9994
      InitPlan 4 (returns $3)
       -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.104..0.182 rows=1 loops=1)
          Hash Cond: (b.team_id = a.country_id)
          -> Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.029..0.049 rows=195 loops=1)
          -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.062..0.063 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              -> Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.030..0.056 rows=1 loops=1)
                 Filter: ((country_name)::text = 'Germany2'::text)
                 Rows Removed by Filter: 194
      InitPlan 9 (returns $8)
       -> Aggregate  (cost=266.50..266.51 rows=1 width=32) (actual time=3.876..3.876 rows=1 loops=1)
          InitPlan 7 (returns $6)
           -> GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=3.651..3.651 rows=1 loops=1)
              Group Key: match_details_1.match_no
              Filter: (count(DISTINCT match_details_1.team_id) = 2)
              Rows Removed by Filter: 2
              InitPlan 5 (returns $4)
               -> Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.045 rows=1 loops=1)
                  Filter: ((country_name)::text = 'Germany1'::text)
                  Rows Removed by Filter: 194
              InitPlan 6 (returns $5)
               -> Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.020..0.045 rows=1 loops=1)
                  Filter: ((country_name)::text = 'Germany2'::text)
                  Rows Removed by Filter: 194
              -> Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=3.620..3.620 rows=4 loops=1)
                  Sort Key: match_details_1.match_no
                  Sort Method: quicksort  Memory: 25kB
                  -> Seq Scan on match_details match_details_1  (cost=0.00..233.97 rows=103 width=10) (actual time=0.069..3.611 rows=4 loops=1)
                     Filter: ((team_id = $4) OR (team_id = $5))
                     Rows Removed by Filter: 9994
          InitPlan 8 (returns $7)
           -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.098..0.189 rows=1 loops=1)
              Hash Cond: (b_1.team_id = a_1.country_id)
              -> Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.025..0.053 rows=195 loops=1)
              -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.063..0.063 rows=1 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 9kB
                  -> Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.058 rows=1 loops=1)
                     Filter: ((country_name)::text = 'Germany2'::text)
                     Rows Removed by Filter: 194
          -> Index Scan using match_no_hash on goal_details  (cost=0.00..8.02 rows=1 width=5) (actual time=3.858..3.860 rows=1 loops=1)
              Index Cond: (match_no = $6)
              Filter: (team_id = $7)
 Planning Time: 0.754 ms
 Execution Time: 13.922 ms
```

Figure 17.b

	QUERY PLAN text
1	Seq Scan on player_mast (cost=549.02..778.02 rows=1 width=9) (actual time=9.096..12.225 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Bitmap Heap Scan on goal_details goal_details_1 (cost=545.00..549.02 rows=1 width=5) (actual time=9.004..9.004 rows=1 loops=1)
6	Recheck Cond: (match_no = $2)
7	Filter: ((team_id = $3) AND (goal_time = $8))
8	Heap Blocks: exact=1
9	InitPlan 3 (returns $2)
10	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.611..4.611 rows=1 loops=1)
11	Group Key: match_details.match_no
12	Filter: (count(DISTINCT match_details.team_id) = 2)
13	Rows Removed by Filter: 2
14	InitPlan 1 (returns $0)
15	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.036..0.072 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany1'::text)
17	Rows Removed by Filter: 194
18	InitPlan 2 (returns $1)
19	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.030 rows=1 loops=1)
20	Filter: ((country_name)::text = 'Germany2'::text)
21	Rows Removed by Filter: 194
22	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.570..4.571 rows=4 loops=1)
23	Sort Key: match_details.match_no
24	Sort Method: quicksort Memory: 25kB
25	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.110..4.550 rows=4 loops=1)
26	Filter: ((team_id = $0) OR (team_id = $1))
27	Rows Removed by Filter: 9994
28	InitPlan 4 (returns $3)
29	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.076..0.113 rows=1 loops=1)
30	Hash Cond: (b.team_id = a.country_id)
31	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.030..0.043 rows=195 loops=1)
32	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.037..0.038 rows=1 loops=1)
33	Buckets: 1024 Batches: 1 Memory Usage: 9kB
34	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.032 rows=1 loops=1)
35	Filter: ((country_name)::text = 'Germany2'::text)
36	Rows Removed by Filter: 194
37	InitPlan 9 (returns $8)
38	-> Aggregate (cost=274.50..274.51 rows=1 width=32) (actual time=4.250..4.250 rows=1 loops=1)
39	InitPlan 7 (returns $6)
40	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=4.115..4.115 rows=1 loops=1)
41	Group Key: match_details_1.match_no
42	Filter: (count(DISTINCT match_details_1.team_id) = 2)
43	Rows Removed by Filter: 2
44	InitPlan 5 (returns $4)
45	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.028 rows=1 loops=1)
46	Filter: ((country_name)::text = 'Germany1'::text)
47	Rows Removed by Filter: 194
48	InitPlan 6 (returns $5)
49	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.043 rows=1 loops=1)
50	Filter: ((country_name)::text = 'Germany2'::text)
52	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=4.089..4.089 rows=4 loops=1)
53	Sort Key: match_details_1.match_no
54	Sort Method: quicksort Memory: 25kB
55	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.040..4.083 rows=4 loops=1)
56	Filter: ((team_id = $4) OR (team_id = $5))
57	Rows Removed by Filter: 9994
58	InitPlan 8 (returns $7)
59	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.074..0.115 rows=1 loops=1)
60	Hash Cond: (b_1.team_id = a_1.country_id)
61	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.022..0.035 rows=195 loops=1)
62	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
63	Buckets: 1024 Batches: 1 Memory Usage: 9kB
64	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.033 rows=1 loops=1)
65	Filter: ((country_name)::text = 'Germany2'::text)
66	Rows Removed by Filter: 194
67	-> Bitmap Heap Scan on goal_details (cost=12.01..16.02 rows=1 width=5) (actual time=4.247..4.247 rows=1 loops=1)
68	Recheck Cond: (match_no = $6)
69	Filter: (team_id = $7)
70	Heap Blocks: exact=1
71	-> Bitmap Index Scan on match_no_gin (cost=0.00..12.01 rows=1 width=0) (actual time=4.127..4.127 rows=1 loops=1)
72	Index Cond: (match_no = $6)
73	-> Bitmap Index Scan on match_no_gin (cost=0.00..12.01 rows=1 width=0) (actual time=4.632..4.632 rows=1 loops=1)
74	Index Cond: (match_no = $2)
75	Planning Time: 0.605 ms
76	Execution Time: 12.439 ms

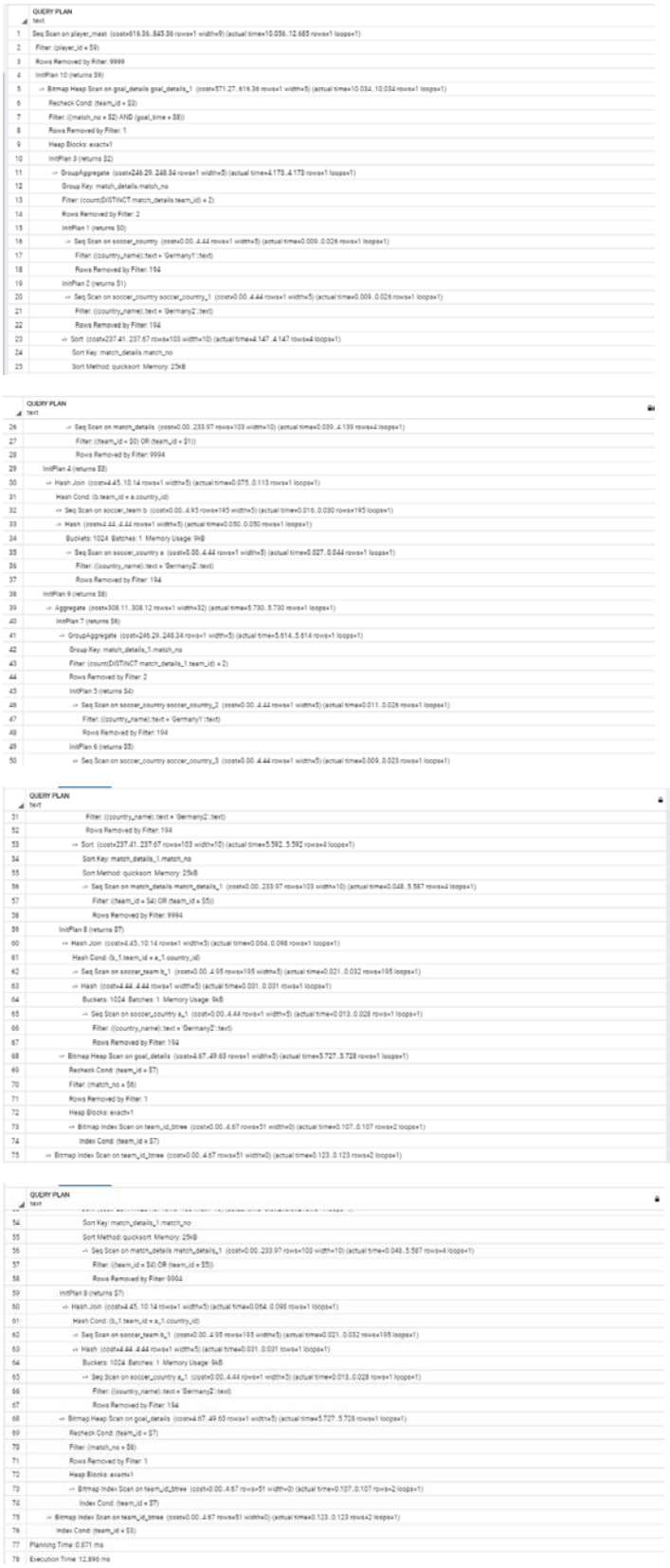
Figure 17.c

c. *Index on goal_details.team_id*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 18.a*
  - *Estimated Cost: 845.36.*
  - *Average Execution Time: 13.15ms.*
- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 18.b*
  - *Estimated Cost: 844.8.*
  - *Average Execution Time: 11.434ms.*
  - *Explanation: Since it gives a time complexity of O (1) which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 18.c.*
  - *Estimated Cost: 852.8.*
  - *Average Execution Time: 13.281ms.*
  - *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, but since there is duplicates, it takes less time and has less cost, but not as much as the hash-based index.



```
QUERY PLAN
text
Seq Scan on player_mast  (cost=616.36..845.36 rows=1 width=9) (actual time=10.056..12.683 rows=1 loops=1)
  Filter: (player_id = $9)
  Rows Removed by Filter: 9999
  InitPlan 10 (returns $9)
    -> Bitmap Heap Scan on goal_details goal_details_1  (cost=571.27..616.36 rows=1 width=5) (actual time=10.034..10.034 rows=1 loops=1)
      Recheck Cond: (team_id = $2)
      Filter: ((match_no = $2) AND (goal_time = $8))
      Rows Removed by Filter: 1
      Heap Blocks: exact=1
      InitPlan 3 (returns $2)
        -> GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=4.173..4.173 rows=1 loops=1)
          Group Key: match_details.match_no
          Filter: (count(DISTINCT match_details.team_id) = 2)
          Rows Removed by Filter: 2
          InitPlan 1 (returns $0)
            -> Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.026 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany1'::text)
              Rows Removed by Filter: 194
          InitPlan 2 (returns $1)
            -> Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.026 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany2'::text)
              Rows Removed by Filter: 194
          -> Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=4.147..4.147 rows=4 loops=1)
            Sort Key: match_details.match_no
            Sort Method: quicksort  Memory: 25kB
```

```
QUERY PLAN
text
            -> Seq Scan on match_details  (cost=0.00..233.97 rows=103 width=10) (actual time=0.039..4.139 rows=4 loops=1)
              Filter: ((team_id = $0) OR (team_id = $1))
              Rows Removed by Filter: 9994
  InitPlan 4 (returns $3)
    -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.075..0.113 rows=1 loops=1)
      Hash Cond: (b.team_id = a.country_id)
      -> Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.016..0.030 rows=195 loops=1)
      -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.050..0.050 rows=1 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        -> Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.027..0.044 rows=1 loops=1)
          Filter: ((country_name)::text = 'Germany2'::text)
          Rows Removed by Filter: 194
  InitPlan 9 (returns $8)
    -> Aggregate  (cost=308.11..308.12 rows=1 width=32) (actual time=5.730..5.730 rows=1 loops=1)
      InitPlan 7 (returns $6)
        -> GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=5.614..5.614 rows=1 loops=1)
          Group Key: match_details_1.match_no
          Filter: (count(DISTINCT match_details_1.team_id) = 2)
          Rows Removed by Filter: 2
          InitPlan 5 (returns $4)
            -> Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.026 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany1'::text)
              Rows Removed by Filter: 194
          InitPlan 6 (returns $5)
            -> Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.023 rows=1 loops=1)
```

```
QUERY PLAN
text
                Filter: ((country_name)::text = 'Germany2'::text)
                Rows Removed by Filter: 194
          -> Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=5.592..5.592 rows=4 loops=1)
            Sort Key: match_details_1.match_no
            Sort Method: quicksort  Memory: 25kB
            -> Seq Scan on match_details match_details_1  (cost=0.00..233.97 rows=103 width=10) (actual time=0.048..5.587 rows=4 loops=1)
              Filter: ((team_id = $4) OR (team_id = $5))
              Rows Removed by Filter: 9994
      InitPlan 8 (returns $7)
        -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.064..0.098 rows=1 loops=1)
          Hash Cond: (b_1.team_id = a_1.country_id)
          -> Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.021..0.032 rows=195 loops=1)
          -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.031..0.031 rows=1 loops=1)
            Buckets: 1024  Batches: 1  Memory Usage: 9kB
            -> Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.028 rows=1 loops=1)
              Filter: ((country_name)::text = 'Germany2'::text)
              Rows Removed by Filter: 194
      -> Bitmap Heap Scan on goal_details  (cost=4.67..49.63 rows=1 width=5) (actual time=5.727..5.728 rows=1 loops=1)
        Recheck Cond: (team_id = $7)
        Filter: (match_no = $6)
        Rows Removed by Filter: 1
        Heap Blocks: exact=1
        -> Bitmap Index Scan on team_id_btree  (cost=0.00..4.67 rows=51 width=0) (actual time=0.107..0.107 rows=2 loops=1)
          Index Cond: (team_id = $7)
    -> Bitmap Index Scan on team_id_btree  (cost=0.00..4.67 rows=51 width=0) (actual time=0.123..0.123 rows=2 loops=1)
      Index Cond: (team_id = $3)
Planning Time: 0.871 ms
Execution Time: 12.896 ms
```

Figure 18.a

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=613.80..844.80 rows=1 width=9) (actual time=11.517..13.291 rows=1 loops=1)
  Filter: (player_id = 59)
  Rows Removed by Filter: 9999
  InitPlan 10 (returns $9)
    ->  Bitmap Heap Scan on goal_details goal_details_1  (cost=570.70..613.80 rows=1 width=8) (actual time=11.493..11.494 rows=1 loops=1)
          Recheck Cond: (team_id = $3)
          Filter: ((match_no = 52) AND (goal_time = 58))
          Rows Removed by Filter: 1
          Heap Blocks: exact=1
          InitPlan 3 (returns $2)
            ->  GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=6.896..6.896 rows=1 loops=1)
                  Group Key: match_details.match_no
                  Filter: (count(DISTINCT match_details.team_id) = 2)
                  Rows Removed by Filter: 2
                  InitPlan 1 (returns $0)
                    ->  Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.071 rows=1 loops=1)
                          Filter: ((country_name)::text = 'Germany1'::text)
                          Rows Removed by Filter: 194
                  InitPlan 2 (returns $1)
                    ->  Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.066 rows=1 loops=1)
                          Filter: ((country_name)::text = 'Germany2'::text)
                          Rows Removed by Filter: 194
                ->  Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=6.855..6.866 rows=4 loops=1)
                      Sort Key: match_details.match_no
                      Sort Method: quicksort  Memory: 25kB
                      ->  Seq Scan on match_details  (cost=0.00..233.97 rows=103 width=10) (actual time=0.092..6.851 rows=4 loops=1)
                            Filter: ((team_id = $0) OR (team_id = $1))
                            Rows Removed by Filter: 9994
    InitPlan 4 (returns $3)
      ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.116..0.207 rows=1 loops=1)
            Hash Cond: (b.team_id = a.country_id)
            ->  Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.018..0.048 rows=195 loops=1)
            ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.083..0.084 rows=1 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 9kB
                  ->  Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.022..0.074 rows=1 loops=1)
                        Filter: ((country_name)::text = 'Germany2'::text)
                        Rows Removed by Filter: 194
    InitPlan 9 (returns $8)
      ->  Aggregate  (cost=307.83..307.84 rows=1 width=32) (actual time=4.361..4.361 rows=1 loops=1)
            InitPlan 7 (returns $6)
              ->  GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=4.232..4.232 rows=1 loops=1)
                    Group Key: match_details_1.match_no
                    Filter: (count(DISTINCT match_details_1.team_id) = 2)
                    Rows Removed by Filter: 2
                    InitPlan 5 (returns $4)
                      ->  Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.024 rows=1 loops=1)
                            Filter: ((country_name)::text = 'Germany1'::text)
                            Rows Removed by Filter: 194
                    InitPlan 6 (returns $5)
                      ->  Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.036 rows=1 loops=1)
                            Filter: ((country_name)::text = 'Germany2'::text)
                            Rows Removed by Filter: 194
                    ->  Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=4.207..4.208 rows=4 loops=1)
                          Sort Key: match_details_1.match_no
                          Sort Method: quicksort  Memory: 25kB
                          ->  Seq Scan on match_details match_details_1  (cost=0.00..233.97 rows=103 width=10) (actual time=0.037..4.201 rows=4 loops=1)
                                Filter: ((team_id = $4) OR (team_id = $5))
                                Rows Removed by Filter: 9994
            InitPlan 8 (returns $7)
              ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.076..0.112 rows=1 loops=1)
                    Hash Cond: (b_1.team_id = a_1.country_id)
                    ->  Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.024..0.035 rows=195 loops=1)
                    ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.036..0.036 rows=1 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.032 rows=1 loops=1)
                                Filter: ((country_name)::text = 'Germany2'::text)
                                Rows Removed by Filter: 194
            ->  Bitmap Heap Scan on goal_details  (cost=4.38..49.35 rows=1 width=5) (actual time=4.357..4.358 rows=1 loops=1)
                  Recheck Cond: (team_id = $7)
                  Filter: (match_no = $6)
                  Rows Removed by Filter: 1
                  Heap Blocks: exact=1
                  ->  Bitmap Index Scan on team_id_hash  (cost=0.00..4.38 rows=51 width=0) (actual time=0.118..0.118 rows=2 loops=1)
                        Index Cond: (team_id = $7)
    ->  Bitmap Index Scan on team_id_hash  (cost=0.00..4.38 rows=51 width=0) (actual time=0.225..0.225 rows=2 loops=1)
          Index Cond: (team_id = $9)
Planning Time: 5.615 ms
Execution Time: 13.609 ms
```

Figure 18.b

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=623.80..852.80 rows=1 width=9) (actual time=10.728..13.518 rows=1 loops=1)
 Filter: (player_id = $9)
 Rows Removed by Filter: 9999
 InitPlan 10 (returns $9)
  ->  Bitmap Heap Scan on goal_details goal_details_1  (cost=578.70..623.80 rows=1 width=5) (actual time=10.686..10.686 rows=1 loops=1)
        Recheck Cond: (team_id = $3)
        Filter: ((match_no = $2) AND (goal_time = $8))
        Rows Removed by Filter: 1
        Heap Blocks: exact=1
        InitPlan 3 (returns $2)
          ->  GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=4.606..4.606 rows=1 loops=1)
                Group Key: match_details.match_no
                Filter: (count(DISTINCT match_details.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 1 (returns $0)
                  ->  Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.043 rows=1 loops=1)
                        Filter: ((country_name)::text = 'Germany1'::text)
                        Rows Removed by Filter: 194
                InitPlan 2 (returns $1)
                  ->  Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.038 rows=1 loops=1)
                        Filter: ((country_name)::text = 'Germany2'::text)
                        Rows Removed by Filter: 194
                ->  Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=4.566..4.566 rows=4 loops=1)
                      Sort Key: match_details.match_no
                      Sort Method: quicksort  Memory: 25kB
```



```
QUERY PLAN
text
                      ->  Seq Scan on match_details  (cost=0.00..233.97 rows=103 width=10) (actual time=0.069..4.544 rows=4 loops=1)
                            Filter: ((team_id = $5) OR (team_id = $1))
                            Rows Removed by Filter: 9994
 InitPlan 4 (returns $3)
  ->  Hash Join  (cost=4.45..10.14 rows=1 width=8) (actual time=0.122..0.185 rows=1 loops=1)
        Hash Cond: (b.team_id = a.country_id)
        ->  Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.030..0.054 rows=195 loops=1)
        ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.068..0.068 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.035..0.061 rows=1 loops=1)
                    Filter: ((country_name)::text = 'Germany2'::text)
                    Rows Removed by Filter: 194
 InitPlan 9 (returns $8)
  ->  Aggregate  (cost=311.83..311.84 rows=1 width=32) (actual time=5.852..5.852 rows=1 loops=1)
        InitPlan 7 (returns $6)
          ->  GroupAggregate  (cost=246.29..248.34 rows=1 width=5) (actual time=5.623..5.623 rows=1 loops=1)
                Group Key: match_details_1.match_no
                Filter: (count(DISTINCT match_details_1.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 5 (returns $4)
                  ->  Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.018..0.065 rows=1 loops=1)
                        Filter: ((country_name)::text = 'Germany1'::text)
                        Rows Removed by Filter: 194
                InitPlan 6 (returns $5)
                  ->  Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.017..0.069 rows=1 loops=1)
```



```
QUERY PLAN
text
                        Filter: ((country_name)::text = 'Germany2'::text)
                        Rows Removed by Filter: 194
                ->  Sort  (cost=237.41..237.67 rows=103 width=10) (actual time=5.534..5.534 rows=4 loops=1)
                      Sort Key: match_details_1.match_no
                      Sort Method: quicksort  Memory: 25kB
                      ->  Seq Scan on match_details match_details_1  (cost=0.00..233.97 rows=103 width=10) (actual time=0.087..5.524 rows=4 loops=1)
                            Filter: ((team_id = $4) OR (team_id = $5))
                            Rows Removed by Filter: 9994
          InitPlan 8 (returns $7)
            ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.120..0.200 rows=1 loops=1)
                  Hash Cond: (b_1.team_id = a_1.country_id)
                  ->  Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.033..0.058 rows=195 loops=1)
                  ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.072..0.072 rows=1 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                        ->  Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.021..0.066 rows=1 loops=1)
                              Filter: ((country_name)::text = 'Germany2'::text)
                              Rows Removed by Filter: 194
        ->  Bitmap Heap Scan on goal_details  (cost=8.38..59.35 rows=1 width=5) (actual time=5.847..5.848 rows=1 loops=1)
              Recheck Cond: (team_id = $7)
              Filter: (match_no = $6)
              Rows Removed by Filter: 1
              Heap Blocks: exact=1
              ->  Bitmap Index Scan on team_id_gin  (cost=0.00..8.38 rows=51 width=0) (actual time=0.216..0.216 rows=2 loops=1)
                    Index Cond: (team_id = $7)
  ->  Bitmap Index Scan on team_id_gin  (cost=0.00..8.38 rows=51 width=0) (actual time=0.214..0.214 rows=2 loops=1)
```



```
QUERY PLAN
text
                        Sort Key: match_details_1.match_no
                        Sort Method: quicksort  Memory: 25kB
                        ->  Seq Scan on match_details match_details_1  (cost=0.00..233.97 rows=103 width=10) (actual time=0.087..5.524 rows=4 loops=1)
                              Filter: ((team_id = $4) OR (team_id = $5))
                              Rows Removed by Filter: 9994
          InitPlan 8 (returns $7)
            ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.120..0.200 rows=1 loops=1)
                  Hash Cond: (b_1.team_id = a_1.country_id)
                  ->  Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.033..0.058 rows=195 loops=1)
                  ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.072..0.072 rows=1 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                        ->  Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.021..0.066 rows=1 loops=1)
                              Filter: ((country_name)::text = 'Germany2'::text)
                              Rows Removed by Filter: 194
        ->  Bitmap Heap Scan on goal_details  (cost=8.38..59.35 rows=1 width=5) (actual time=5.847..5.848 rows=1 loops=1)
              Recheck Cond: (team_id = $7)
              Filter: (match_no = $6)
              Rows Removed by Filter: 1
              Heap Blocks: exact=1
              ->  Bitmap Index Scan on team_id_gin  (cost=0.00..8.38 rows=51 width=0) (actual time=0.216..0.216 rows=2 loops=1)
                    Index Cond: (team_id = $7)
  ->  Bitmap Index Scan on team_id_gin  (cost=0.00..8.38 rows=51 width=0) (actual time=0.214..0.214 rows=2 loops=1)
        Index Cond: (team_id = $3)
Planning Time: 3.213 ms
Execution Time: 13.930 ms
```

Figure 18.c

d. *Index on match_details.team_id*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 19.a*
  - *Estimated Cost: 723.24.*
  - *Average Execution Time: 6.51ms.*
- *Explanation: The reason behind this is that the B-Tree index runs with time complexity O (log n), which is faster than O (n) without any indices.*
- *Hash-based Index:*
  - *Query Plan: Figure 19.b*
  - *Estimated Cost: 722.1.*
  - *Average Execution Time: 5.991ms.*
  - *Explanation: The reason behind this is that the hash-based index runs with time complexity O (1), which is much faster than O (n) without any indices.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 19.c.*
  - *Estimated Cost: 738.08.*
  - *Average Execution Time: 6.664ms.*
  - *Explanation: The reason behind this is that the bitmap index runs faster than O (n) without any indices because of the bit-wise operation that it does.*
- *Best Scenario on the Column*

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, but since there is duplicates, it takes less time and has less cost, but not as much as the hash-based index.

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=494.24..723.24 rows=1 width=9) (actual time=2.534..5.660 rows=1 loops=1)
  Filter: (player_id = $9)
  Rows Removed by Filter: 9999
  InitPlan 10 (returns $9)
    -> Seq Scan on goal_details goal_details_1  (cost=364.74..494.24 rows=1 width=8) (actual time=1.557..2.519 rows=1 loops=1)
         Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
         Rows Removed by Filter: 4999
         InitPlan 3 (returns $2)
           -> GroupAggregate  (cost=111.67..113.72 rows=1 width=5) (actual time=0.121..0.121 rows=1 loops=1)
                Group Key: match_details.match_no
                Filter: (count(DISTINCT match_details.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 1 (returns $0)
                  -> Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.029 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany1'::text)
                       Rows Removed by Filter: 194
                InitPlan 2 (returns $1)
                  -> Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany2'::text)
                       Rows Removed by Filter: 194
                -> Sort  (cost=102.80..103.06 rows=103 width=10) (actual time=0.105..0.105 rows=4 loops=1)
                     Sort Key: match_details.match_no
                     Sort Method: quicksort  Memory: 25kB
                     -> Bitmap Heap Scan on match_details  (cost=9.40..99.36 rows=103 width=10) (actual time=0.095..0.096 rows=4 loops=1)
                          Recheck Cond: ((team_id = $0) OR (team_id = $1))
                          Heap Blocks: exact=1
                          -> BitmapOr  (cost=9.40..9.40 rows=103 width=0) (actual time=0.092..0.092 rows=0 loops=1)
                               -> Bitmap Index Scan on team_idm_btree  (cost=0.00..4.67 rows=52 width=0) (actual time=0.063..0.063 rows=2 loops=1)
                                    Index Cond: (team_id = $0)
                               -> Bitmap Index Scan on team_idm_btree  (cost=0.00..4.67 rows=52 width=0) (actual time=0.029..0.029 rows=2 loops=1)
                                    Index Cond: (team_id = $1)
  InitPlan 4 (returns $3)
    -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.043..0.079 rows=1 loops=1)
         Hash Cond: (b.team_id = a.country_id)
         -> Seq Scan on soccer_team b  (cost=0.00..4.93 rows=193 width=3) (actual time=0.010..0.023 rows=193 loops=1)
         -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.028..0.028 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              -> Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
                   Filter: ((country_name)::text = 'Germany2'::text)
                   Rows Removed by Filter: 194
  InitPlan 9 (returns $8)
    -> Aggregate  (cost=240.86..240.87 rows=1 width=32) (actual time=1.343..1.343 rows=1 loops=1)
         InitPlan 7 (returns $6)
           -> GroupAggregate  (cost=111.67..113.72 rows=1 width=5) (actual time=0.070..0.070 rows=1 loops=1)
                Group Key: match_details_1.match_no
                Filter: (count(DISTINCT match_details_1.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 5 (returns $4)
                  -> Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.023 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany1'::text)
                       Rows Removed by Filter: 194
                InitPlan 6 (returns $5)
                  -> Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.023 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany2'::text)
                       Rows Removed by Filter: 194
                -> Sort  (cost=102.80..103.06 rows=103 width=10) (actual time=0.060..0.060 rows=4 loops=1)
                     Sort Key: match_details_1.match_no
                     Sort Method: quicksort  Memory: 25kB
                     -> Bitmap Heap Scan on match_details match_details_1  (cost=9.40..99.36 rows=103 width=10) (actual time=0.057..0.057 rows=4 loops=1)
                          Recheck Cond: ((team_id = $4) OR (team_id = $5))
                          Heap Blocks: exact=1
                          -> BitmapOr  (cost=9.40..9.40 rows=103 width=0) (actual time=0.052..0.052 rows=0 loops=1)
                               -> Bitmap Index Scan on team_idm_btree  (cost=0.00..4.67 rows=52 width=0) (actual time=0.026..0.027 rows=2 loops=1)
                                    Index Cond: (team_id = $4)
                               -> Bitmap Index Scan on team_idm_btree  (cost=0.00..4.67 rows=52 width=0) (actual time=0.025..0.025 rows=2 loops=1)
                                    Index Cond: (team_id = $5)
  InitPlan 8 (returns $7)
    -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.039..0.071 rows=1 loops=1)
         Hash Cond: (b_1.team_id = a_1.country_id)
         -> Seq Scan on soccer_team b_1  (cost=0.00..4.93 rows=193 width=3) (actual time=0.009..0.019 rows=193 loops=1)
         -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.027..0.027 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              -> Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.023 rows=1 loops=1)
                   Filter: ((country_name)::text = 'Germany2'::text)
                   Rows Removed by Filter: 194
  -> Seq Scan on goal_details  (cost=0.00..117.00 rows=1 width=3) (actual time=0.152..1.340 rows=1 loops=1)
       Filter: ((match_no = $6) AND (team_id = $7))
       Rows Removed by Filter: 4999
Planning Time: 2.580 ms
Execution Time: 5.832 ms
```

Figure 19.a

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=493.10..722.10 rows=1 width=9) (actual time=2.630..5.790 rows=1 loops=1)
  Filter: (player_id = $9)
  Rows Removed by Filter: 9999
  InitPlan 10 (returns $9)
    -> Seq Scan on goal_details goal_details_1  (cost=363.60..493.10 rows=1 width=5) (actual time=1.583..2.627 rows=1 loops=1)
         Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
         Rows Removed by Filter: 4999
         InitPlan 3 (returns $2)
           -> GroupAggregate  (cost=111.10..113.15 rows=1 width=5) (actual time=0.317..0.317 rows=1 loops=1)
                Group Key: match_details.match_no
                Filter: (count(DISTINCT match_details.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 1 (returns $0)
                  -> Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.019..0.071 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany1'::text)
                       Rows Removed by Filter: 194
                InitPlan 2 (returns $1)
                  -> Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.032..0.113 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany2'::text)
                       Rows Removed by Filter: 194
                -> Sort  (cost=102.23..102.49 rows=103 width=10) (actual time=0.266..0.267 rows=4 loops=1)
                     Sort Key: match_details.match_no
                     Sort Method: quicksort  Memory: 25kB
                     -> Bitmap Heap Scan on match_details  (cost=8.83..98.70 rows=103 width=10) (actual time=0.244..0.247 rows=4 loops=1)
                          Recheck Cond: ((team_id = $0) OR (team_id = $1))
                          Heap Blocks: exact=1
                          -> BitmapOr  (cost=8.83..8.83 rows=103 width=0) (actual time=0.235..0.235 rows=0 loops=1)
                               -> Bitmap Index Scan on team_idm_hash  (cost=0.00..4.39 rows=52 width=0) (actual time=0.102..0.102 rows=2 loops=1)
                                    Index Cond: (team_id = $0)
                               -> Bitmap Index Scan on team_idm_hash  (cost=0.00..4.39 rows=52 width=0) (actual time=0.130..0.130 rows=2 loops=1)
                                    Index Cond: (team_id = $1)
  InitPlan 4 (returns $3)
    -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.071..0.113 rows=1 loops=1)
         Hash Cond: (b.team_id = a.country_id)
         -> Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.024..0.038 rows=195 loops=1)
         -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.036..0.037 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              -> Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.016..0.033 rows=1 loops=1)
                   Filter: ((country_name)::text = 'Germany2'::text)
                   Rows Removed by Filter: 194
  InitPlan 9 (returns $8)
    -> Aggregate  (cost=240.29..240.30 rows=1 width=32) (actual time=1.133..1.133 rows=1 loops=1)
         InitPlan 7 (returns $6)
           -> GroupAggregate  (cost=111.10..113.15 rows=1 width=5) (actual time=0.080..0.080 rows=1 loops=1)
                Group Key: match_details_1.match_no
                Filter: (count(DISTINCT match_details_1.team_id) = 2)
                Rows Removed by Filter: 2
                InitPlan 5 (returns $4)
                  -> Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany1'::text)
                       Rows Removed by Filter: 194
                InitPlan 6 (returns $5)
                  -> Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
                       Filter: ((country_name)::text = 'Germany2'::text)
                       Rows Removed by Filter: 194
                -> Sort  (cost=102.23..102.49 rows=103 width=10) (actual time=0.069..0.069 rows=4 loops=1)
                     Sort Key: match_details_1.match_no
                     Sort Method: quicksort  Memory: 25kB
                     -> Bitmap Heap Scan on match_details match_details_1  (cost=8.83..98.79 rows=103 width=10) (actual time=0.063..0.065 rows=4 loops=1)
                          Recheck Cond: ((team_id = $4) OR (team_id = $5))
                          Heap Blocks: exact=1
                          -> BitmapOr  (cost=8.83..8.83 rows=103 width=0) (actual time=0.061..0.061 rows=0 loops=1)
                               -> Bitmap Index Scan on team_idm_hash  (cost=0.00..4.39 rows=52 width=0) (actual time=0.032..0.032 rows=2 loops=1)
                                    Index Cond: (team_id = $4)
                               -> Bitmap Index Scan on team_idm_hash  (cost=0.00..4.39 rows=52 width=0) (actual time=0.028..0.029 rows=2 loops=1)
                                    Index Cond: (team_id = $5)
  InitPlan 8 (returns $7)
    -> Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.047..0.085 rows=1 loops=1)
         Hash Cond: (b_1.team_id = a_1.country_id)
         -> Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.011..0.023 rows=195 loops=1)
         -> Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.030..0.030 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              -> Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
                   Filter: ((country_name)::text = 'Germany2'::text)
                   Rows Removed by Filter: 194
  -> Seq Scan on goal_details  (cost=0.00..117.00 rows=1 width=5) (actual time=0.176..1.130 rows=1 loops=1)
       Filter: ((match_no = $6) AND (team_id = $7))
       Rows Removed by Filter: 4999
Planning Time: 2.790 ms
Execution Time: 6.050 ms
```

Figure 19.b

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=509.08..738.08 rows=1 width=9) (actual time=4.818..7.558 rows=1 loops=1)
  Filter: (player_id = $9)
  Rows Removed by Filter: 9999
  InitPlan 10 (returns $9)
    ->  Seq Scan on goal_details goal_details_1  (cost=379.58..509.08 rows=1 width=5) (actual time=2.681..4.798 rows=1 loops=1)
          Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
          Rows Removed by Filter: 4999
          InitPlan 3 (returns $2)
            ->  GroupAggregate  (cost=119.10..121.14 rows=1 width=5) (actual time=0.113..0.113 rows=1 loops=1)
                  Group Key: match_details.match_no
                  Filter: (count(DISTINCT match_details.team_id) = 2)
                  Rows Removed by Filter: 2
                  InitPlan 1 (returns $0)
                    ->  Seq Scan on soccer_country  (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.032 rows=1 loops=1)
                          Filter: ((country_name)::text = 'Germany1'::text)
                          Rows Removed by Filter: 194
                  InitPlan 2 (returns $1)
                    ->  Seq Scan on soccer_country soccer_country_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
                          Filter: ((country_name)::text = 'Germany2'::text)
                          Rows Removed by Filter: 194
            ->  Sort  (cost=110.22..110.48 rows=103 width=10) (actual time=0.093..0.094 rows=4 loops=1)
                  Sort Key: match_details.match_no
                  Sort Method: quicksort  Memory: 25kB
                  ->  Bitmap Heap Scan on match_details  (cost=16.82..106.78 rows=103 width=10) (actual time=0.082..0.083 rows=4 loops=1)
                        Recheck Cond: ((team_id = $0) OR (team_id = $1))
                        Heap Blocks: exact=1
                        ->  BitmapOr  (cost=16.82..16.82 rows=103 width=0) (actual time=0.079..0.079 rows=0 loops=1)
                              ->  Bitmap Index Scan on team_idm_gin  (cost=0.00..8.39 rows=52 width=0) (actual time=0.047..0.047 rows=2 loops=1)
                                    Index Cond: (team_id = $0)
                              ->  Bitmap Index Scan on team_idm_gin  (cost=0.00..8.39 rows=52 width=0) (actual time=0.031..0.031 rows=2 loops=1)
                                    Index Cond: (team_id = $1)
    InitPlan 4 (returns $3)
      ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.089..0.130 rows=1 loops=1)
            Hash Cond: (b.team_id = a.country_id)
            ->  Seq Scan on soccer_team b  (cost=0.00..4.95 rows=195 width=5) (actual time=0.012..0.026 rows=195 loops=1)
            ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 9kB
                  ->  Seq Scan on soccer_country a  (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.032 rows=1 loops=1)
                        Filter: ((country_name)::text = 'Germany2'::text)
                        Rows Removed by Filter: 194
    InitPlan 9 (returns $8)
      ->  Aggregate  (cost=248.29..248.30 rows=1 width=32) (actual time=2.417..2.417 rows=1 loops=1)
            InitPlan 7 (returns $6)
              ->  GroupAggregate  (cost=119.10..121.14 rows=1 width=5) (actual time=0.175..0.175 rows=1 loops=1)
                    Group Key: match_details_1.match_no
                    Filter: (count(DISTINCT match_details_1.team_id) = 2)
                    Rows Removed by Filter: 2
                    InitPlan 5 (returns $4)
                      ->  Seq Scan on soccer_country soccer_country_2  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
                            Filter: ((country_name)::text = 'Germany1'::text)
                            Rows Removed by Filter: 194
                    InitPlan 6 (returns $5)
                      ->  Seq Scan on soccer_country soccer_country_3  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.039 rows=1 loops=1)
                            Filter: ((country_name)::text = 'Germany2'::text)
                            Rows Removed by Filter: 194
                    ->  Sort  (cost=110.22..110.48 rows=103 width=10) (actual time=0.099..0.100 rows=4 loops=1)
                          Sort Key: match_details_1.match_no
                          Sort Method: quicksort  Memory: 25kB
                          ->  Bitmap Heap Scan on match_details match_details_1  (cost=16.82..106.78 rows=103 width=10) (actual time=0.093..0.094 rows=4 loops=1)
                                Recheck Cond: ((team_id = $4) OR (team_id = $5))
                                Heap Blocks: exact=1
                                ->  BitmapOr  (cost=16.82..16.82 rows=103 width=0) (actual time=0.091..0.091 rows=0 loops=1)
                                      ->  Bitmap Index Scan on team_idm_gin  (cost=0.00..8.39 rows=52 width=0) (actual time=0.033..0.033 rows=2 loops=1)
                                            Index Cond: (team_id = $4)
                                      ->  Bitmap Index Scan on team_idm_gin  (cost=0.00..8.39 rows=52 width=0) (actual time=0.056..0.056 rows=2 loops=1)
                                            Index Cond: (team_id = $5)
          InitPlan 8 (returns $7)
            ->  Hash Join  (cost=4.45..10.14 rows=1 width=5) (actual time=0.058..0.135 rows=1 loops=1)
                  Hash Cond: (b_1.team_id = a_1.country_id)
                  ->  Seq Scan on soccer_team b_1  (cost=0.00..4.95 rows=195 width=5) (actual time=0.012..0.029 rows=195 loops=1)
                  ->  Hash  (cost=4.44..4.44 rows=1 width=5) (actual time=0.038..0.038 rows=1 loops=1)
                        Buckets: 1024  Batches: 1  Memory Usage: 9kB
                        ->  Seq Scan on soccer_country a_1  (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.035 rows=1 loops=1)
                              Filter: ((country_name)::text = 'Germany2'::text)
                              Rows Removed by Filter: 194
    ->  Seq Scan on goal_details  (cost=0.00..117.00 rows=1 width=5) (actual time=0.313..2.410 rows=1 loops=1)
          Filter: ((match_no = $6) AND (team_id = $7))
          Rows Removed by Filter: 4999
Planning Time: 1.223 ms
Execution Time: 7.984 ms
```

Figure 19.c

f. Index on soccer_country.country_name

- *Flags Modifications: The flags enable_seqscan, enable_hashjoin, enable_mergejoin were set off.*
- *Without Indices:*
  - *Query Plan: Figure (xiii).*
  - *Estimated Cost: 130000000995.83.*
  - *Average Execution Time: 13.892ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 20.a*
  - *Estimated Cost: 700000001018.18.*
  - *Average Execution Time: 12.793ms.*
- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster, and also there is no duplicates in this column in this table which makes it even faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 20.b*
  - *Estimated Cost: 700000001017.31.*
  - *Average Execution Time: 12.71ms.*
  - *Explanation: Since it gives a time complexity of O (1) which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 20.c.*
  - *Estimated Cost: 700000001041.33.*
  - *Average Execution Time: 12.99ms.*

- *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster, but it is not working that efficient with columns that has no duplicates, so it is not so much fast as applying B-tree or hash-based index on this column.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and it is faster compared to B-Tree index or bitmap (GIN) index since it has a time complexity of O (1) but the B-Tree index has a time complexity of O (log n), and the time complexity of bitmap index is O (n) when having no duplicates, which is the case with this column, but it is faster due to bit-wise operation, but search is what takes O (n).

	QUERY PLAN text
1	Seq Scan on player_mast (cost=130000000766.83..130000000995.83 rows=1 width=9) (actual time=8.757..12.482 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=120000000637.33..120000000766.83 rows=1 width=5) (actual time=7.455..8.738 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=3.069..3.069 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.017..0.035 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.048..3.049 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.053..3.037 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Nested Loop (cost=20000000000.00..20000000011.83 rows=1 width=5) (actual time=0.027..0.185 rows=1 loops=1)
29	Join Filter: (a.country_id = b.team_id)
30	Rows Removed by Join Filter: 194
31	-> Seq Scan on soccer_country a (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.014..0.051 rows=1 loops=1)
32	Filter: ((country_name)::text = 'Germany2'::text)
33	Rows Removed by Filter: 194
34	-> Seq Scan on soccer_team b (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.011..0.036 rows=195 loops=1)
35	InitPlan 9 (returns $8)
36	-> Aggregate (cost=60000000377.16..60000000377.17 rows=1 width=32) (actual time=4.183..4.183 rows=1 loops=1)
37	InitPlan 7 (returns $6)
38	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=2.985..2.985 rows=1 loops=1)
39	Group Key: match_details_1.match_no
40	Filter: (count(DISTINCT match_details_1.team_id) = 2)
41	Rows Removed by Filter: 2
42	InitPlan 5 (returns $4)
43	-> Seq Scan on soccer_country soccer_country_2 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
44	Filter: ((country_name)::text = 'Germany1'::text)
45	Rows Removed by Filter: 194
46	InitPlan 6 (returns $5)
47	-> Seq Scan on soccer_country soccer_country_3 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.012..0.029 rows=1 loops=1)
48	Filter: ((country_name)::text = 'Germany2'::text)
49	Rows Removed by Filter: 194
50	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.969..2.969 rows=4 loops=1)
51	Sort Key: match_details_1.match_no
52	Sort Method: quicksort Memory: 25kB
53	-> Seq Scan on match_details match_details_1 (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.040..2.963 rows=4 loops=1)
54	Filter: ((team_id = $4) OR (team_id = $5))
55	Rows Removed by Filter: 9994
56	InitPlan 8 (returns $7)
57	-> Nested Loop (cost=20000000000.00..20000000011.83 rows=1 width=5) (actual time=0.024..0.081 rows=1 loops=1)
58	Join Filter: (a_1.country_id = b_1.team_id)
59	Rows Removed by Join Filter: 194
60	-> Seq Scan on soccer_country a_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
61	Filter: ((country_name)::text = 'Germany2'::text)
62	Rows Removed by Filter: 194
63	-> Seq Scan on soccer_team b_1 (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.010..0.022 rows=195 loops=1)
64	-> Seq Scan on goal_details (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=3.081..4.180 rows=1 loops=1)
65	Filter: ((match_no = $6) AND (team_id = $7))
66	Rows Removed by Filter: 4999
67	Planning Time: 47.547 ms
68	Execution Time: 12.619 ms

Figure (xiii)

	QUERY PLAN text
1	Seq Scan on player_mast (cost=70000000789.18..70000001018.18 rows=1 width=9) (actual time=11.281..15.128 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=60000000659.68..60000000789.18 rows=1 width=5) (actual time=9.181..11.257 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=10000000253.74..10000000255.79 rows=1 width=5) (actual time=3.407..3.407 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Index Scan using country_name_btree on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.018..0.019 rows=1 loops=1)
15	Index Cond: ((country_name)::text = 'Germany1'::text)
16	InitPlan 2 (returns $1)
17	-> Index Scan using country_name_btree on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.005..0.005 rows=1 loops=1)
18	Index Cond: ((country_name)::text = 'Germany2'::text)
19	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.358..3.358 rows=4 loops=1)
20	Sort Key: match_details.match_no
21	Sort Method: quicksort Memory: 25kB
22	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.035..3.349 rows=4 loops=1)
23	Filter: ((team_id = $0) OR (team_id = $1))
24	Rows Removed by Filter: 9994
25	InitPlan 4 (returns $3)
26	-> Nested Loop (cost=10000000000.15..10000000015.55 rows=1 width=5) (actual time=0.073..0.116 rows=1 loops=1)
27	Join Filter: (a.country_id = b.team_id)
28	Rows Removed by Join Filter: 194
29	-> Index Scan using country_name_btree on soccer_country a (cost=0.14..8.16 rows=1 width=5) (actual time=0.036..0.037 rows=1 loops=1)
30	Index Cond: ((country_name)::text = 'Germany2'::text)
31	-> Seq Scan on soccer_team b (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.033..0.047 rows=195 loops=1)
32	InitPlan 9 (returns $8)
33	-> Aggregate (cost=30000000388.34..30000000388.35 rows=1 width=32) (actual time=5.636..5.636 rows=1 loops=1)
34	InitPlan 7 (returns $6)
35	-> GroupAggregate (cost=10000000253.74..10000000255.79 rows=1 width=5) (actual time=2.878..2.878 rows=1 loops=1)
36	Group Key: match_details_1.match_no
37	Filter: (count(DISTINCT match_details_1.team_id) = 2)
38	Rows Removed by Filter: 2
39	InitPlan 5 (returns $4)
40	-> Index Scan using country_name_btree on soccer_country soccer_country_2 (cost=0.14..8.16 rows=1 width=5) (actual time=0.012..0.012 rows=1 loops=1)
41	Index Cond: ((country_name)::text = 'Germany1'::text)
42	InitPlan 6 (returns $5)
43	-> Index Scan using country_name_btree on soccer_country soccer_country_3 (cost=0.14..8.16 rows=1 width=5) (actual time=0.004..0.004 rows=1 loops=1)
44	Index Cond: ((country_name)::text = 'Germany2'::text)
45	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.850..2.851 rows=4 loops=1)
46	Sort Key: match_details_1.match_no
47	Sort Method: quicksort Memory: 25kB
48	-> Seq Scan on match_details match_details_1 (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.033..2.845 rows=4 loops=1)
49	Filter: ((team_id = $4) OR (team_id = $5))
50	Rows Removed by Filter: 9994
51	InitPlan 8 (returns $7)
52	-> Nested Loop (cost=10000000000.15..10000000015.55 rows=1 width=5) (actual time=0.104..0.194 rows=1 loops=1)
53	Join Filter: (a_1.country_id = b_1.team_id)
54	Rows Removed by Join Filter: 194
55	-> Index Scan using country_name_btree on soccer_country a_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.077..0.078 rows=1 loops=1)
56	Index Cond: ((country_name)::text = 'Germany2'::text)
57	-> Seq Scan on soccer_team b_1 (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.023..0.045 rows=195 loops=1)
58	-> Seq Scan on goal_details (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=3.091..5.630 rows=1 loops=1)
59	Filter: ((match_no = $6) AND (team_id = $7))
60	Rows Removed by Filter: 4999
61	Planning Time: 0.952 ms
62	Execution Time: 15.303 ms

Figure 20.a

```
1  Seq Scan on player_mast  (cost=70000000788.31..70000001017.31 rows=1 width=9) (actual time=11.125..13.591 rows=1 loops=1)
2    Filter: (player_id = $9)
3    Rows Removed by Filter: 9999
4    InitPlan 10 (returns $9)
5      ->  Seq Scan on goal_details goal_details_1  (cost=60000000658.81..60000000788.31 rows=1 width=5) (actual time=9.394..11.105 rows=1 loops=1)
6            Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7            Rows Removed by Filter: 4999
8            InitPlan 3 (returns $2)
9              ->  GroupAggregate  (cost=10000000253.45..10000000255.50 rows=1 width=5) (actual time=3.392..3.393 rows=1 loops=1)
10               Group Key: match_details.match_no
11               Filter: (count(DISTINCT match_details.team_id) = 2)
12               Rows Removed by Filter: 2
13               InitPlan 1 (returns $0)
14                 ->  Index Scan using country_name_hash on soccer_country  (cost=0.00..8.02 rows=1 width=5) (actual time=0.008..0.009 rows=1 loops=1)
15                       Index Cond: ((country_name)::text = 'Germany1'::text)
16               InitPlan 2 (returns $1)
17                 ->  Index Scan using country_name_hash on soccer_country soccer_country_1  (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
18                       Index Cond: ((country_name)::text = 'Germany2'::text)
19               ->  Sort  (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.360..3.361 rows=4 loops=1)
20                     Sort Key: match_details.match_no
21                     Sort Method: quicksort  Memory: 25kB
22                     ->  Seq Scan on match_details  (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.021..3.352 rows=4 loops=1)
23                           Filter: ((team_id = $0) OR (team_id = $1))
24                           Rows Removed by Filter: 9994
25           InitPlan 4 (returns $3)

26             ->  Nested Loop  (cost=10000000000.00..10000000015.41 rows=1 width=5) (actual time=0.045..0.089 rows=1 loops=1)
27                   Join Filter: (a.country_id = b.team_id)
28                   Rows Removed by Join Filter: 194
29                   ->  Index Scan using country_name_hash on soccer_country a  (cost=0.00..8.02 rows=1 width=5) (actual time=0.016..0.017 rows=1 loops=1)
30                         Index Cond: ((country_name)::text = 'Germany2'::text)
31                   ->  Seq Scan on soccer_team b  (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.026..0.040 rows=195 loops=1)
32           InitPlan 9 (returns $8)
33             ->  Aggregate  (cost=30000000387.90..30000000387.91 rows=1 width=32) (actual time=5.894..5.894 rows=1 loops=1)
34                   InitPlan 7 (returns $6)
35                     ->  GroupAggregate  (cost=10000000253.45..10000000255.50 rows=1 width=5) (actual time=3.584..3.584 rows=1 loops=1)
36                           Group Key: match_details_1.match_no
37                           Filter: (count(DISTINCT match_details_1.team_id) = 2)
38                           Rows Removed by Filter: 2
39                           InitPlan 5 (returns $4)
40                             ->  Index Scan using country_name_hash on soccer_country soccer_country_2  (cost=0.00..8.02 rows=1 width=5) (actual time=0.006..0.007 rows=1 loops=1)
41                                   Index Cond: ((country_name)::text = 'Germany1'::text)
42                           InitPlan 6 (returns $5)
43                             ->  Index Scan using country_name_hash on soccer_country soccer_country_3  (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
44                                   Index Cond: ((country_name)::text = 'Germany2'::text)
45                           ->  Sort  (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.533..3.534 rows=4 loops=1)
46                                 Sort Key: match_details_1.match_no
47                                 Sort Method: quicksort  Memory: 25kB
48                                 ->  Seq Scan on match_details match_details_1  (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.021..3.527 rows=4 loops=1)
49                                       Filter: ((team_id = $4) OR (team_id = $5))
50                                       Rows Removed by Filter: 9994

51                 InitPlan 8 (returns $7)
52                   ->  Nested Loop  (cost=10000000000.00..10000000015.41 rows=1 width=5) (actual time=0.055..0.151 rows=1 loops=1)
53                         Join Filter: (a_1.country_id = b_1.team_id)
54                         Rows Removed by Join Filter: 194
55                         ->  Index Scan using country_name_hash on soccer_country a_1  (cost=0.00..8.02 rows=1 width=5) (actual time=0.019..0.020 rows=1 loops=1)
56                               Index Cond: ((country_name)::text = 'Germany2'::text)
57                         ->  Seq Scan on soccer_team b_1  (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.032..0.056 rows=195 loops=1)
58                   ->  Seq Scan on goal_details  (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=3.755..5.888 rows=1 loops=1)
59                         Filter: ((match_no = $6) AND (team_id = $7))
60                         Rows Removed by Filter: 4999
61   Planning Time: 0.536 ms
62   Execution Time: 13.706 ms
```

Figure 20.b

```
QUERY PLAN
text
Seq Scan on player_mast  (cost=70000000812.33..70000001041.33 rows=1 width=9) (actual time=10.561..13.416 rows=1 loops=1)
 Filter: (player_id = $9)
 Rows Removed by Filter: 9999
 InitPlan 10 (returns $9)
  -> Seq Scan on goal_details goal_details_1  (cost=60000000682.83..60000000812.33 rows=1 width=5) (actual time=9.260..10.540 rows=1 loops=1)
     Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
     Rows Removed by Filter: 4999
     InitPlan 3 (returns $2)
      -> GroupAggregate  (cost=10000000261.45..10000000263.50 rows=1 width=5) (actual time=3.314..3.314 rows=1 loops=1)
         Group Key: match_details.match_no
         Filter: (count(DISTINCT match_details.team_id) = 2)
         Rows Removed by Filter: 2
         InitPlan 1 (returns $0)
          -> Bitmap Heap Scan on soccer_country  (cost=8.01..12.02 rows=1 width=5) (actual time=0.030..0.030 rows=1 loops=1)
             Recheck Cond: ((country_name)::text = 'Germany1'::text)
             Heap Blocks: exact=1
             -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.025..0.025 rows=1 loops=1)
                Index Cond: ((country_name)::text = 'Germany1'::text)
         InitPlan 2 (returns $1)
          -> Bitmap Heap Scan on soccer_country soccer_country_1  (cost=8.01..12.02 rows=1 width=5) (actual time=0.005..0.005 rows=1 loops=1)
             Recheck Cond: ((country_name)::text = 'Germany2'::text)
             Heap Blocks: exact=1
             -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.004..0.005 rows=1 loops=1)
                Index Cond: ((country_name)::text = 'Germany2'::text)
         -> Sort  (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.284..3.284 rows=4 loops=1)
            Sort Key: match_details.match_no
            Sort Method: quicksort  Memory: 25kB
            -> Seq Scan on match_details  (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.047..3.269 rows=4 loops=1)
               Filter: ((team_id = $0) OR (team_id = $1))
               Rows Removed by Filter: 9994
     InitPlan 4 (returns $3)
      -> Nested Loop  (cost=10000000008.01..10000000019.41 rows=1 width=5) (actual time=0.054..0.097 rows=1 loops=1)
         Join Filter: (a.country_id = b.team_id)
         Rows Removed by Join Filter: 194
         -> Bitmap Heap Scan on soccer_country a  (cost=8.01..12.02 rows=1 width=5) (actual time=0.023..0.023 rows=1 loops=1)
            Recheck Cond: ((country_name)::text = 'Germany2'::text)
            Heap Blocks: exact=1
            -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.019..0.019 rows=1 loops=1)
               Index Cond: ((country_name)::text = 'Germany2'::text)
         -> Seq Scan on soccer_team b  (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.028..0.042 rows=195 loops=1)
     InitPlan 9 (returns $8)
      -> Aggregate  (cost=30000000399.91..30000000399.92 rows=1 width=32) (actual time=5.828..5.828 rows=1 loops=1)
         InitPlan 7 (returns $6)
          -> GroupAggregate  (cost=10000000261.45..10000000263.50 rows=1 width=5) (actual time=4.274..4.274 rows=1 loops=1)
             Group Key: match_details_1.match_no
             Filter: (count(DISTINCT match_details_1.team_id) = 2)
             Rows Removed by Filter: 2
             InitPlan 5 (returns $4)
              -> Bitmap Heap Scan on soccer_country soccer_country_2  (cost=8.01..12.02 rows=1 width=5) (actual time=0.007..0.007 rows=1 loops=1)
                 Recheck Cond: ((country_name)::text = 'Germany1'::text)
                 Heap Blocks: exact=1
                 -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.006..0.006 rows=1 loops=1)
                    Index Cond: ((country_name)::text = 'Germany1'::text)
             InitPlan 6 (returns $5)
              -> Bitmap Heap Scan on soccer_country soccer_country_3  (cost=8.01..12.02 rows=1 width=5) (actual time=0.004..0.005 rows=1 loops=1)
                 Recheck Cond: ((country_name)::text = 'Germany2'::text)
                 Heap Blocks: exact=1
                 -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.004..0.004 rows=1 loops=1)
                    Index Cond: ((country_name)::text = 'Germany2'::text)
             -> Sort  (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.226..4.227 rows=4 loops=1)
                Sort Key: match_details_1.match_no
                Sort Method: quicksort  Memory: 25kB
                -> Seq Scan on match_details match_details_1  (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.022..4.220 rows=4 loops=1)
                   Filter: ((team_id = $4) OR (team_id = $5))
                   Rows Removed by Filter: 9994
         InitPlan 8 (returns $7)
          -> Nested Loop  (cost=10000000008.01..10000000019.41 rows=1 width=5) (actual time=0.089..0.178 rows=1 loops=1)
             Join Filter: (a_1.country_id = b_1.team_id)
             Rows Removed by Join Filter: 194
             -> Bitmap Heap Scan on soccer_country a_1  (cost=8.01..12.02 rows=1 width=5) (actual time=0.037..0.037 rows=1 loops=1)
                Recheck Cond: ((country_name)::text = 'Germany2'::text)
                Heap Blocks: exact=1
                -> Bitmap Index Scan on country_name_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.031..0.031 rows=1 loops=1)
                   Index Cond: ((country_name)::text = 'Germany2'::text)
             -> Seq Scan on soccer_team b_1  (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.046..0.067 rows=195 loops=1)
         -> Seq Scan on soccer_team b_1  (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.046..0.067 rows=195 loops=1)
        -> Seq Scan on goal_details  (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=4.478..5.823 rows=1 loops=1)
           Filter: ((match_no = $6) AND (team_id = $7))
           Rows Removed by Filter: 4999
Planning Time: 0.996 ms
Execution Time: 13.706 ms
```

Figure 20.c

g. *Index on soccer_country.country_id*

- *Flags Modifications: The flags enable_seqscan, enable_hashjoin, enable_mergejoin were set off.*
- *Without Indices:*
  - *Query Plan: Figure (xiii).*
  - *Estimated Cost: 1100000001020.07.*
  - *Average Execution Time: 13.892ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 21.a*
  - *Estimated Cost: 700000001018.18.*
  - *Average Execution Time: 10.864ms.*
- *Explanation: Since it gives a time complexity of O (log n) which makes the query finishes executing faster, and also there is no duplicates in this column in this table which makes it even faster.*
- *Hash-based Index:*
  - *Query Plan: Figure 21.b*
  - *Estimated Cost: 1100000001041.78.*
  - *Average Execution Time: 13.75ms.*
  - *Explanation: Since it gives a time complexity of O (1) which makes the query finishes executing extremely fast.*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 21.c.*
  - *Estimated Cost: 1100000002570.86.*
  - *Average Execution Time: 17.86ms.*

- *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster, but it is not working that efficient with columns that has no duplicates, so it is not so much fast as applying B-tree or hash-based index on this column.*

- *Best Scenario on the Column*

  Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the B-Tree index on this column, since it has the least execution time, and the least cost. Even though the time complexity of the hash-based index is faster than B-Tree index but there are other query types (aggregate functions), the B-Tree works faster.

	QUERY PLAN text
1	Seq Scan on player_mast (cost=110000000791.07..110000001020.07 rows=1 width=9) (actual time=8.855..10.467 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=100000000661.57..100000000791.07 rows=1 width=5) (actual time=7.873..8.799 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=4.088..4.088 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.021..0.088 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.025..0.082 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.056..4.056 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.127..4.035 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Nested Loop (cost=10000000000.15..10000000023.95 rows=1 width=5) (actual time=0.037..0.103 rows=1 loops=1)
29	Join Filter: (a.country_id = b.team_id)
30	Rows Removed by Join Filter: 194
31	-> Index Scan using country_id_btree on soccer_country a (cost=0.14..16.56 rows=1 width=5) (actual time=0.012..0.041 rows=1 loops=1)
32	Filter: ((country_name)::text = 'Germany2'::text)
33	Rows Removed by Filter: 194
34	-> Seq Scan on soccer_team b (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.023..0.034 rows=195 loops=1)
35	InitPlan 9 (returns $8)
36	-> Aggregate (cost=50000000389.28..50000000389.29 rows=1 width=32) (actual time=3.629..3.629 rows=1 loops=1)
37	InitPlan 7 (returns $6)
38	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=2.631..2.631 rows=1 loops=1)
39	Group Key: match_details_1.match_no
40	Filter: (count(DISTINCT match_details_1.team_id) = 2)
41	Rows Removed by Filter: 2
42	InitPlan 5 (returns $4)
43	-> Seq Scan on soccer_country soccer_country_2 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.010..0.025 rows=1 loops=1)
44	Filter: ((country_name)::text = 'Germany1'::text)
45	Rows Removed by Filter: 194
46	InitPlan 6 (returns $5)
47	-> Seq Scan on soccer_country soccer_country_3 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.011..0.025 rows=1 loops=1)
48	Filter: ((country_name)::text = 'Germany2'::text)
49	Rows Removed by Filter: 194
50	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.620..2.620 rows=4 loops=1)
51	Sort Key: match_details_1.match_no
52	Sort Method: quicksort Memory: 25kB
53	-> Seq Scan on match_details match_details_1 (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.037..2.618 rows=4 loops=1)
54	Filter: ((team_id = $4) OR (team_id = $5))
55	Rows Removed by Filter: 9994
56	InitPlan 8 (returns $7)
57	-> Nested Loop (cost=10000000000.15..10000000023.95 rows=1 width=5) (actual time=0.015..0.076 rows=1 loops=1)
58	Join Filter: (a_1.country_id = b_1.team_id)
59	Rows Removed by Join Filter: 194
60	-> Index Scan using country_id_btree on soccer_country a_1 (cost=0.14..16.56 rows=1 width=5) (actual time=0.004..0.030 rows=1 loops=1)
61	Filter: ((country_name)::text = 'Germany2'::text)
62	Rows Removed by Filter: 194
63	-> Seq Scan on soccer_team b_1 (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.009..0.019 rows=195 loops=1)
64	-> Seq Scan on goal_details (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=2.722..3.626 rows=1 loops=1)
65	Filter: ((match_no = $6) AND (team_id = $7))
66	Rows Removed by Filter: 4999
67	Planning Time: 0.912 ms
68	Execution Time: 10.672 ms

Figure 21.a

	QUERY PLAN text
1	Seq Scan on player_mast (cost=110000000812.78..110000001041.78 rows=1 width=9) (actual time=9.326..11.459 rows=1 loops=1)
2	Filter: (player_id = $11)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $11)
5	-> Seq Scan on goal_details goal_details_1 (cost=100000000683.28..100000000812.78 rows=1 width=5) (actual time=8.215..9.273 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $4) AND (goal_time = $10))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=3.247..3.247 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.019..0.059 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.013..0.029 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=3.225..3.226 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.100..3.212 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $4)
28	-> Nested Loop (cost=10000000000.00..10000000034.80 rows=1 width=5) (actual time=0.034..0.242 rows=1 loops=1)
29	-> Seq Scan on soccer_team b (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.020..0.033 rows=195 loops=1)
30	-> Index Scan using country_id_hash on soccer_country a (cost=0.00..0.14 rows=1 width=5) (actual time=0.001..0.001 rows=0 loops=195)
31	Index Cond: (country_id = b.team_id)
32	Filter: ((country_name)::text = 'Germany2'::text)
33	Rows Removed by Filter: 1
34	InitPlan 9 (returns $10)
35	-> Aggregate (cost=50000000400.14..50000000400.15 rows=1 width=32) (actual time=4.689..4.689 rows=1 loops=1)
36	InitPlan 7 (returns $7)
37	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=2.668..2.669 rows=1 loops=1)
38	Group Key: match_details_1.match_no
39	Filter: (count(DISTINCT match_details_1.team_id) = 2)
40	Rows Removed by Filter: 2
41	InitPlan 5 (returns $5)
42	-> Seq Scan on soccer_country soccer_country_2 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.010..0.025 rows=1 loops=1)
43	Filter: ((country_name)::text = 'Germany1'::text)
44	Rows Removed by Filter: 194
45	InitPlan 6 (returns $6)
46	-> Seq Scan on soccer_country soccer_country_3 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
47	Filter: ((country_name)::text = 'Germany2'::text)
48	Rows Removed by Filter: 194
49	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=2.657..2.657 rows=4 loops=1)
50	Sort Key: match_details_1.match_no
51	Sort Method: quicksort Memory: 25kB
52	-> Seq Scan on match_details match_details_1 (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.037..2.653 rows=4 loops=1)
53	Filter: ((team_id = $5) OR (team_id = $6))
54	Rows Removed by Filter: 9994
55	InitPlan 8 (returns $9)
56	-> Nested Loop (cost=10000000000.00..10000000034.80 rows=1 width=5) (actual time=0.088..0.449 rows=1 loops=1)
57	-> Seq Scan on soccer_team b_1 (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.023..0.062 rows=195 loops=1)
58	-> Index Scan using country_id_hash on soccer_country a_1 (cost=0.00..0.14 rows=1 width=5) (actual time=0.001..0.001 rows=0 loops=195)
59	Index Cond: (country_id = b_1.team_id)
60	Filter: ((country_name)::text = 'Germany2'::text)
61	Rows Removed by Filter: 1
62	-> Seq Scan on goal_details (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=3.134..4.685 rows=1 loops=1)
63	Filter: ((match_no = $7) AND (team_id = $9))
64	Rows Removed by Filter: 4999
65	Planning Time: 0.640 ms
66	Execution Time: 11.870 ms

Figure 21.b

	QUERY PLAN text
1	Seq Scan on player_mast (cost=1100000002341.86..1100000002570.86 rows=1 width=9) (actual time=14.934..16.697 rows=1 loops=1)
2	Filter: (player_id = $11)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $11)
5	-> Seq Scan on goal_details goal_details_1 (cost=100000002212.36..100000002341.86 rows=1 width=5) (actual time=13.989..14.917 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $4) AND (goal_time = $10))
7	Rows Removed by Filter: 4999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=4.693..4.693 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.023..0.047 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.014..0.048 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.664..4.665 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.064..4.655 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $4)
28	-> Nested Loop (cost=10000000000.05..10000000799.34 rows=1 width=5) (actual time=0.060..1.636 rows=1 loops=1)
29	-> Seq Scan on soccer_team b (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.022..0.036 rows=195 loops=1)
30	-> Bitmap Heap Scan on soccer_country a (cost=0.05..4.06 rows=1 width=5) (actual time=0.002..0.002 rows=0 loops=195)
31	Recheck Cond: (country_id = b.team_id)
32	Filter: ((country_name)::text = 'Germany2'::text)
33	Rows Removed by Filter: 1
34	Heap Blocks: exact=195
35	-> Bitmap Index Scan on country_id_gin (cost=0.00..0.05 rows=1 width=0) (actual time=0.001..0.001 rows=1 loops=195)
36	Index Cond: (country_id = b.team_id)
37	InitPlan 9 (returns $10)
38	-> Aggregate (cost=500000001164.67..500000001164.68 rows=1 width=32) (actual time=7.641..7.641 rows=1 loops=1)
39	InitPlan 7 (returns $7)
40	-> GroupAggregate (cost=30000000246.29..30000000248.34 rows=1 width=5) (actual time=4.106..4.106 rows=1 loops=1)
41	Group Key: match_details_1.match_no
42	Filter: (count(DISTINCT match_details_1.team_id) = 2)
43	Rows Removed by Filter: 2
44	InitPlan 5 (returns $5)
45	-> Seq Scan on soccer_country soccer_country_2 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.013..0.028 rows=1 loops=1)
46	Filter: ((country_name)::text = 'Germany1'::text)
47	Rows Removed by Filter: 194
48	InitPlan 6 (returns $6)
49	-> Seq Scan on soccer_country soccer_country_3 (cost=10000000000.00..10000000004.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
50	Filter: ((country_name)::text = 'Germany2'::text)
51	Rows Removed by Filter: 194
52	-> Sort (cost=10000000237.41..10000000237.67 rows=103 width=10) (actual time=4.085..4.086 rows=4 loops=1)
53	Sort Key: match_details_1.match_no
54	Sort Method: quicksort Memory: 25kB
55	-> Seq Scan on match_details match_details_1 (cost=10000000000.00..10000000233.97 rows=103 width=10) (actual time=0.039..4.081 rows=4 loops=1)
56	Filter: ((team_id = $5) OR (team_id = $6))
57	Rows Removed by Filter: 9994
58	InitPlan 8 (returns $9)
59	-> Nested Loop (cost=10000000000.05..10000000799.34 rows=1 width=5) (actual time=0.043..2.335 rows=1 loops=1)
60	-> Seq Scan on soccer_team b_1 (cost=10000000000.00..10000000004.95 rows=195 width=5) (actual time=0.016..0.062 rows=195 loops=1)
61	-> Bitmap Heap Scan on soccer_country a_1 (cost=0.05..4.06 rows=1 width=5) (actual time=0.004..0.004 rows=0 loops=195)
62	Recheck Cond: (country_id = b_1.team_id)
63	Filter: ((country_name)::text = 'Germany2'::text)
64	Rows Removed by Filter: 1
65	Heap Blocks: exact=195
66	-> Bitmap Index Scan on country_id_gin (cost=0.00..0.05 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=195)
67	Index Cond: (country_id = b_1.team_id)
68	-> Seq Scan on goal_details (cost=10000000000.00..10000000117.00 rows=1 width=5) (actual time=6.459..7.636 rows=1 loops=1)
69	Filter: ((match_no = $7) AND (team_id = $9))
70	Rows Removed by Filter: 4999
71	Planning Time: 0.651 ms
72	Execution Time: 16.872 ms

Figure 21.c

h. *Index on goal_details.goal_time*

- *Flags Modifications: All flags were set on.*
- *Without Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *B-Tree Index:*
  - *Query Plan: Figure 22.a*
  - *Estimated Cost: 821.27.*
  - *Average Execution Time: 11.664ms.*
- *Explanation: The B-Tree index has a time complexity of O (log n) and it also works well with aggregate functions.*
- *Hash-based Index:*
  - *Query Plan: Figure 22.b*
  - *Estimated Cost: 820.77.*
  - *Average Execution Time: 11.034ms.*
  - *Explanation: Since it gives a time complexity of O (1).*
- *Bitmap (GIN) Index:*
  - *Query Plan: Figure 22.c.*
  - *Estimated Cost: 878.99.*
  - *Average Execution Time: 12.02ms.*
  - *Explanation: Since it gives a fast time complexity due to the bit-wise operation of its vector which makes the query finishes executing faster.*
- *Best Scenario on the Column*

Referring to the execution time, and the cost of every case, we can infer that the best scenario is using the hash-based index on this column, since it has the least execution time, and the least cost. Even though the B-Tree index is very efficient with aggregate functions, but the hash-based index is still faster.

	QUERY PLAN text
1	Seq Scan on player_mast (cost=642.27..871.27 rows=1 width=9) (actual time=6.876..8.479 rows=1 loops=1)
2	Filter: (player_id = $10)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $10)
5	-> Index Scan using goal_time_btree on goal_details goal_details_1 (cost=634.25..642.27 rows=1 width=5) (actual time=6.847..6.848 rows=1 loops=1)
6	Index Cond: (goal_time = $9)
7	Filter: ((match_no = $2) AND (team_id = $3))
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.018..3.018 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.025 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.001..3.002 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.040..2.997 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.048..0.081 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.013..0.024 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.030..0.030 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.027 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $9)
37	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=3.732..3.732 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=2.699..2.699 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 2
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.027 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194
47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.023 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=2.685..2.685 rows=4 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 25kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.040..2.678 rows=4 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9994
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.044..0.078 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.010..0.022 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.028..0.028 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=2.789..3.730 rows=1 loops=1)
67	Filter: ((match_no = $6) AND (team_id = $7))
68	Rows Removed by Filter: 4999
69	Planning Time: 0.455 ms
70	Execution Time: 8.622 ms

Figure 22.a

	QUERY PLAN text
1	Seq Scan on player_mast (cost=641.99..870.99 rows=1 width=9) (actual time=8.578..12.114 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Index Scan using goal_time_hash on goal_details goal_details_1 (cost=633.97..641.99 rows=1 width=5) (actual time=8.542..8.544 rows=1 loops=1)
6	Index Cond: (goal_time = $8)
7	Filter: ((match_no = $2) AND (team_id = $3))
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.141..3.141 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 2
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.031 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.103..3.104 rows=4 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 25kB
24	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.055..3.094 rows=4 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9994
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.120..0.314 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.033..0.177 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.074..0.074 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.020..0.065 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $8)
37	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=5.054..5.054 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.195..3.195 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 2
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.033 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194
47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.169..3.169 rows=4 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 25kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.055..3.155 rows=4 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9994
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.065..0.100 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.020..0.033 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.032..0.032 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.027 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=3.312..5.048 rows=1 loops=1)
67	Filter: ((match_no = $6) AND (team_id = $7))
68	Rows Removed by Filter: 4999
69	Planning Time: 2.747 ms
70	Execution Time: 12.292 ms

Figure 22.b

	QUERY PLAN text
1	Seq Scan on player_mast (cost=649.99..878.99 rows=1 width=9) (actual time=8.008..9.887 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Bitmap Heap Scan on goal_details goal_details_1 (cost=645.97..649.99 rows=1 width=5) (actual time=7.991..7.992 rows=1 loops=1)
6	Recheck Cond: (goal_time = $8)
7	Filter: ((match_no = $2) AND (team_id = $3))
8	Heap Blocks: exact=1
9	InitPlan 3 (returns $2)
10	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=3.941..3.942 rows=1 loops=1)
11	Group Key: match_details.match_no
12	Filter: (count(DISTINCT match_details.team_id) = 2)
13	Rows Removed by Filter: 2
14	InitPlan 1 (returns $0)
15	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
16	Filter: ((country_name)::text = 'Germany1'::text)
17	Rows Removed by Filter: 194
18	InitPlan 2 (returns $1)
19	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.028 rows=1 loops=1)
20	Filter: ((country_name)::text = 'Germany2'::text)
21	Rows Removed by Filter: 194
22	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=3.912..3.912 rows=4 loops=1)
23	Sort Key: match_details.match_no
24	Sort Method: quicksort Memory: 25kB
25	-> Seq Scan on match_details (cost=0.00..233.97 rows=103 width=10) (actual time=0.042..3.904 rows=4 loops=1)

	QUERY PLAN text
26	Filter: ((team_id = $0) OR (team_id = $1))
27	Rows Removed by Filter: 9994
28	InitPlan 4 (returns $3)
29	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.110..0.151 rows=1 loops=1)
30	Hash Cond: (b.team_id = a.country_id)
31	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.025..0.037 rows=195 loops=1)
32	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.043..0.043 rows=1 loops=1)
33	Buckets: 1024 Batches: 1 Memory Usage: 9kB
34	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.020..0.037 rows=1 loops=1)
35	Filter: ((country_name)::text = 'Germany2'::text)
36	Rows Removed by Filter: 194
37	InitPlan 9 (returns $8)
38	-> Aggregate (cost=375.48..375.49 rows=1 width=32) (actual time=3.872..3.872 rows=1 loops=1)
39	InitPlan 7 (returns $6)
40	-> GroupAggregate (cost=246.29..248.34 rows=1 width=5) (actual time=2.800..2.800 rows=1 loops=1)
41	Group Key: match_details_1.match_no
42	Filter: (count(DISTINCT match_details_1.team_id) = 2)
43	Rows Removed by Filter: 2
44	InitPlan 5 (returns $4)
45	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.029 rows=1 loops=1)
46	Filter: ((country_name)::text = 'Germany1'::text)
47	Rows Removed by Filter: 194
48	InitPlan 6 (returns $5)
49	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
50	Filter: ((country_name)::text = 'Germany2'::text)

	QUERY PLAN text
49	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.009..0.024 rows=1 loops=1)
50	Filter: ((country_name)::text = 'Germany2'::text)
51	Rows Removed by Filter: 194
52	-> Sort (cost=237.41..237.67 rows=103 width=10) (actual time=2.783..2.784 rows=4 loops=1)
53	Sort Key: match_details_1.match_no
54	Sort Method: quicksort Memory: 25kB
55	-> Seq Scan on match_details match_details_1 (cost=0.00..233.97 rows=103 width=10) (actual time=0.045..2.773 rows=4 loops=1)
56	Filter: ((team_id = $4) OR (team_id = $5))
57	Rows Removed by Filter: 9994
58	InitPlan 8 (returns $7)
59	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.048..0.083 rows=1 loops=1)
60	Hash Cond: (b_1.team_id = a_1.country_id)
61	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.011..0.024 rows=195 loops=1)
62	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.029..0.029 rows=1 loops=1)
63	Buckets: 1024 Batches: 1 Memory Usage: 9kB
64	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.025 rows=1 loops=1)
65	Filter: ((country_name)::text = 'Germany2'::text)
66	Rows Removed by Filter: 194
67	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=2.895..3.869 rows=1 loops=1)
68	Filter: ((match_no = $6) AND (team_id = $7))
69	Rows Removed by Filter: 4999
70	-> Bitmap Index Scan on goal_time_gin (cost=0.00..12.01 rows=1 width=0) (actual time=3.892..3.892 rows=1 loops=1)
71	Index Cond: (goal_time = $8)
72	Planning Time: 2.676 ms
73	Execution Time: 10.045 ms

Figure 22.c

i. *Multiple Indices*

- *Flags Modifications: All flags were set on.*
- *Before Indices:*
  - *Query Plan: Figure (xii).*
  - *Estimated Cost: 992.47.*
  - *Average Execution Time: 14.853ms.*
- *After Indices:*
  - *Indices Used: Hash-based indices on goal_details.goal_time, player_mast.player_id, and match_details.team_id.*
  - *Query Plan: Figure 23.*
  - *Estimated Cost: 379.64.*
  - *Average Execution Time: 1.944ms.*
  - *Explanation: Since the hash-based index has a time complexity of O (1), so when it is applied on those columns, the query became much faster.*

j. *Best case applied on the query*

According to the costs and execution time of all cases applied on the columns, the best-case scenario is to apply the multiple indices case on the query since it has the least execution time and cost, as it uses the hash-based index when searching for exact values, which gives a very fast cost and execution time compared to the B-Tree index and the bitmap index.

	QUERY PLAN text
1	Index Scan using player_hash on player_mast (cost=371.62..379.64 rows=1 width=9) (actual time=1.416..1.416 rows=1 loops=1)
2	Index Cond: (player_id = $9)
3	InitPlan 10 (returns $9)
4	-> Index Scan using goal_time_hash on goal_details goal_details_1 (cost=363.60..371.62 rows=1 width=5) (actual time=1.404..1.405 rows=1 loops=1)
5	Index Cond: (goal_time = 58)
6	Filter: ((match_no = $2) AND (team_id = $3))
7	InitPlan 3 (returns $2)
8	-> GroupAggregate (cost=111.10..113.15 rows=1 width=3) (actual time=0.082..0.082 rows=1 loops=1)
9	Group Key: match_details.match_no
10	Filter: (count(DISTINCT match_details.team_id) = 2)
11	Rows Removed by Filter: 2
12	InitPlan 1 (returns $0)
13	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=3) (actual time=0.012..0.029 rows=1 loops=1)
14	Filter: ((country_name)::text = 'Germany1'::text)
15	Rows Removed by Filter: 194
16	InitPlan 2 (returns $1)
17	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
18	Filter: ((country_name)::text = 'Germany2'::text)
19	Rows Removed by Filter: 194
20	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.069..0.070 rows=4 loops=1)
21	Sort Key: match_details.match_no
22	Sort Method: quicksort Memory: 25kB
23	-> Bitmap Heap Scan on match_details (cost=8.83..98.79 rows=103 width=10) (actual time=0.054..0.065 rows=4 loops=1)
24	Recheck Cond: ((team_id = $0) OR (team_id = $1))
25	Heap Blocks: exact=1

	QUERY PLAN text
26	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.061..0.061 rows=0 loops=1)
27	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.032..0.032 rows=2 loops=1)
28	Index Cond: (team_id = $0)
29	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.029..0.029 rows=2 loops=1)
30	Index Cond: (team_id = $1)
31	InitPlan 4 (returns $3)
32	-> Hash Join (cost=4.45..10.14 rows=1 width=3) (actual time=0.046..0.085 rows=1 loops=1)
33	Hash Cond: (b.team_id = a.country_id)
34	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.011..0.024 rows=195 loops=1)
35	-> Hash (cost=4.44..4.44 rows=1 width=3) (actual time=0.029..0.030 rows=1 loops=1)
36	Buckets: 1024 Batches: 1 Memory Usage: 9kB
37	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
38	Filter: ((country_name)::text = 'Germany2'::text)
39	Rows Removed by Filter: 194
40	InitPlan 9 (returns $8)
41	-> Aggregate (cost=240.29..240.30 rows=1 width=32) (actual time=1.221..1.221 rows=1 loops=1)
42	InitPlan 7 (returns $6)
43	-> GroupAggregate (cost=111.10..113.15 rows=1 width=5) (actual time=0.122..0.122 rows=1 loops=1)
44	Group Key: match_details_1.match_no
45	Filter: (count(DISTINCT match_details_1.team_id) = 2)
46	Rows Removed by Filter: 2
47	InitPlan 5 (returns $4)
48	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.015..0.034 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany1'::text)
50	Rows Removed by Filter: 194

	QUERY PLAN text
51	InitPlan 0 (returns $5)
52	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.027 rows=1 loops=1)
53	Filter: ((country_name)::text = 'Germany2'::text)
54	Rows Removed by Filter: 194
55	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.104..0.104 rows=4 loops=1)
56	Sort Key: match_details_1.match_no
57	Sort Method: quicksort Memory: 25kB
58	-> Bitmap Heap Scan on match_details match_details_1 (cost=8.83..98.79 rows=103 width=10) (actual time=0.090..0.092 rows=4 loops=1)
59	Recheck Cond: ((team_id = $4) OR (team_id = $5))
60	Heap Blocks: exact=1
61	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.085..0.085 rows=0 loops=1)
62	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.054..0.054 rows=2 loops=1)
63	Index Cond: (team_id = $4)
64	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.031..0.031 rows=2 loops=1)
65	Index Cond: (team_id = $5)
66	InitPlan 8 (returns $7)
67	-> Hash Join (cost=4.45..10.14 rows=1 width=3) (actual time=0.050..0.090 rows=1 loops=1)
68	Hash Cond: (b_1.team_id = a_1.country_id)
69	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.012..0.027 rows=195 loops=1)
70	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.031..0.031 rows=1 loops=1)
71	Buckets: 1024 Batches: 1 Memory Usage: 9kB
72	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
73	Filter: ((country_name)::text = 'Germany2'::text)
74	Rows Removed by Filter: 194
75	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=0.229..1.218 rows=1 loops=1)

	QUERY PLAN text
55	-> Sort (cost=102.23..102.49 rows=103 width=10) (actual time=0.104..0.104 rows=4 loops=1)
56	Sort Key: match_details_1.match_no
57	Sort Method: quicksort Memory: 25kB
58	-> Bitmap Heap Scan on match_details match_details_1 (cost=8.83..98.79 rows=103 width=10) (actual time=0.090..0.092 rows=4 loops=1)
59	Recheck Cond: ((team_id = $4) OR (team_id = $5))
60	Heap Blocks: exact=1
61	-> BitmapOr (cost=8.83..8.83 rows=103 width=0) (actual time=0.085..0.085 rows=0 loops=1)
62	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.054..0.054 rows=2 loops=1)
63	Index Cond: (team_id = $4)
64	-> Bitmap Index Scan on team_idm_hash (cost=0.00..4.39 rows=52 width=0) (actual time=0.031..0.031 rows=2 loops=1)
65	Index Cond: (team_id = $5)
66	InitPlan 8 (returns $7)
67	-> Hash Join (cost=4.45..10.14 rows=1 width=5) (actual time=0.050..0.090 rows=1 loops=1)
68	Hash Cond: (b_1.team_id = a_1.country_id)
69	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.012..0.027 rows=195 loops=1)
70	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.031..0.031 rows=1 loops=1)
71	Buckets: 1024 Batches: 1 Memory Usage: 9kB
72	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.027 rows=1 loops=1)
73	Filter: ((country_name)::text = 'Germany2'::text)
74	Rows Removed by Filter: 194
75	-> Seq Scan on goal_details (cost=0.00..117.00 rows=1 width=5) (actual time=0.229..1.218 rows=1 loops=1)
76	Filter: ((match_no = $6) AND (team_id = $7))
77	Rows Removed by Filter: 4999
78	Planning Time: 7.031 ms
79	Execution Time: 1.606 ms

Figure 23