

Pushpak Sharma

Following

Nov 19, 2022 · 8 min read · Listen



Save



# SQL Query Performance Optimization

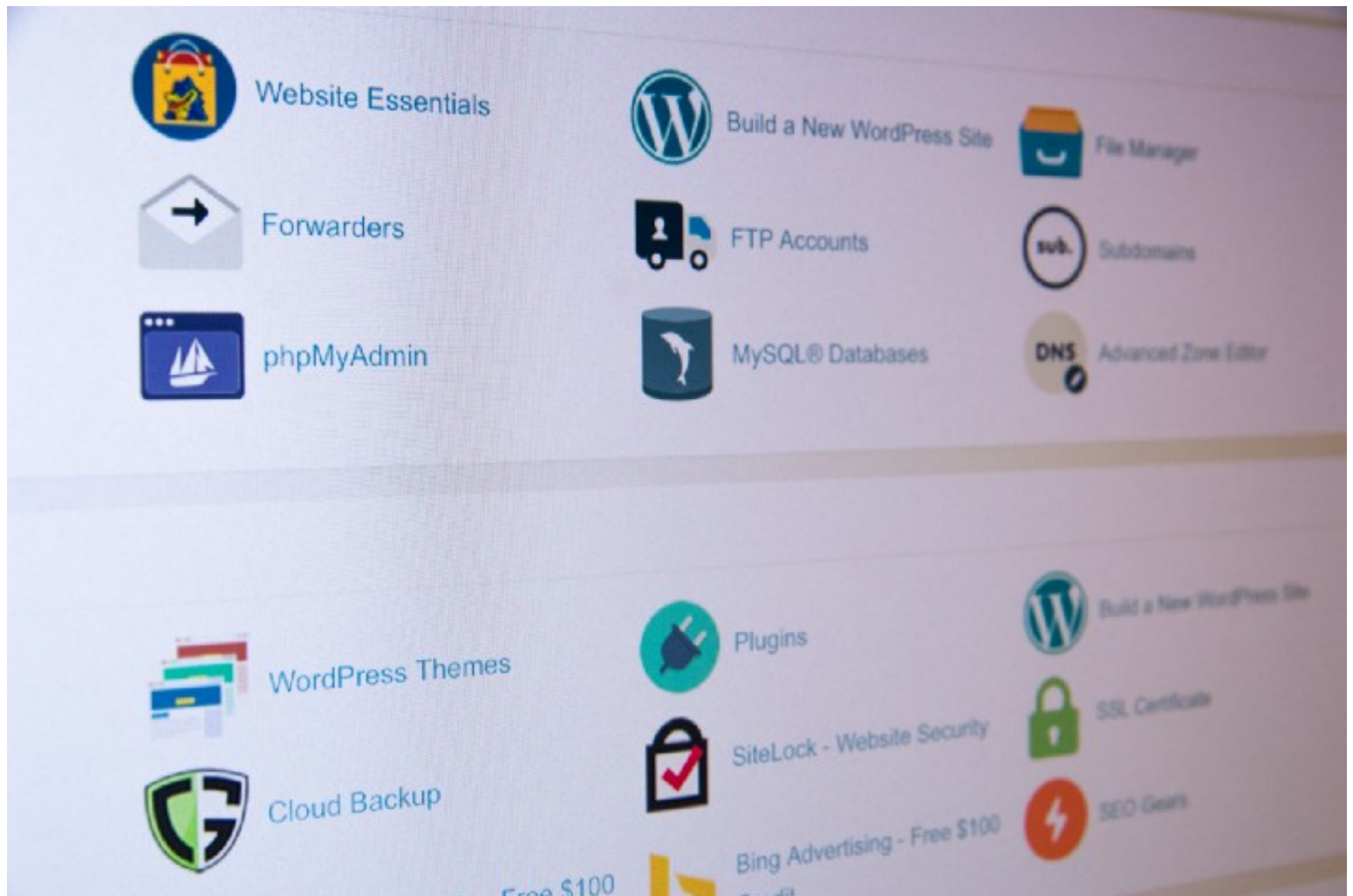


Photo by [Stephen Phillips - Hostreviews.co.uk](#) on [Unsplash](#)

## Objective

- Bitmap and hash indexes
- Using different types of indexes to improve performance
- Challenges with joining tables



150



1



- When to use partitioning to improve performance
- Collecting statistics about data in tables

### What is scanning?

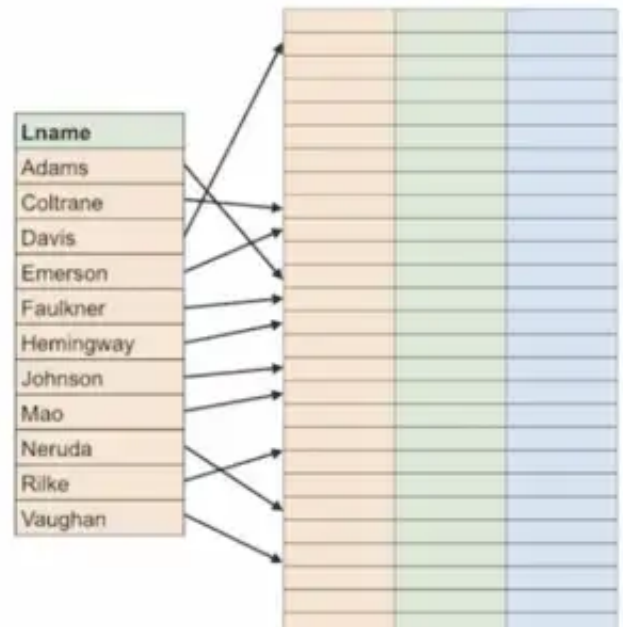
Scanning is a **linear operation**, moving from one row to the next, and performing some operation, like applying a filter to implement a WHERE clause. Scanning is simple.

The database fetches data blocks from **persistent storage or cache**, **applies a filter, join, or other operation on the row**, and **then moves on to the next row**. The time it takes to finish the scan is based on the number of rows in the table.

### How do Indexes save scanning?

## Indexes Save Scanning

- Indexes are ordered
- Faster to search index for an attribute value
- Points to location of row
- Example: filter by checking index for match, then retrieve row



### Types of Indexes

# Types of Indexes

- B-tree, for equality and range queries
- Hash indexes, for equality
- Bitmap, for inclusion
- Specialized indexes, for geo-spatial or user-defined indexing strategies

## Types of Join

1

### **Nested Loop Join**

Compare all rows in both tables to each other.

2

### **Hash Join**

Calculate hash value of key and join based on matching hash values.

3

### **Sort Merge Join**

Sort both tables and then join rows while taking advantage of order.

# Nested Loop Joins

- Loop through one table
- For each row, loop through the other table
- At each step, compare keys
- Simple to implement
- Can be expensive

# Hash Joins

- Compute hash values of key values in smaller table
- Store in hash table, which has hash value and row attributes
- Scan larger table; find rows from smaller hash table

# Sort Merge Joins

- Sort both tables
- Compare rows like nested loop join, but ...
- Stop when it is not possible to find a match later in the table because of the sort order
- Scan the driving table only once

## What is Partitioning

One way to **avoid scanning large amounts of data** is to break those large amounts of data into smaller pieces.

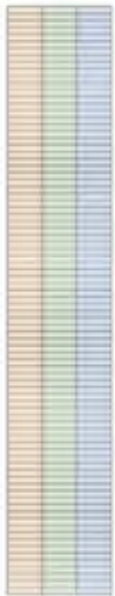
This is the basic idea behind partitioning.

Large tables are stored as a set of smaller tables. This not only helps with query performance, but it can improve the speed of data loads and some delete operations.

# What Is Partitioning?

- Storing table data in multiple sub-tables, known as partitions
- Used to improve query, load, and delete operations
- Used for large tables
- When subset of data is accessed or changed
- Can be expensive

## Large Tables = Large Indexes

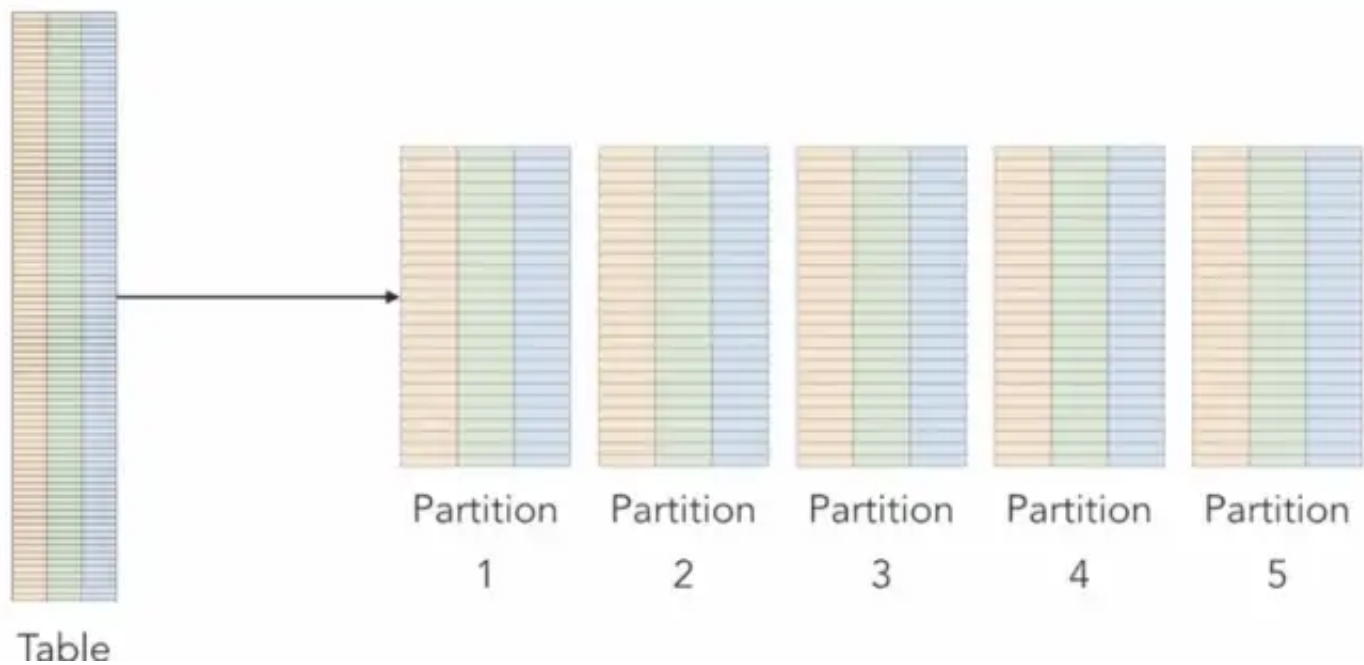


Table

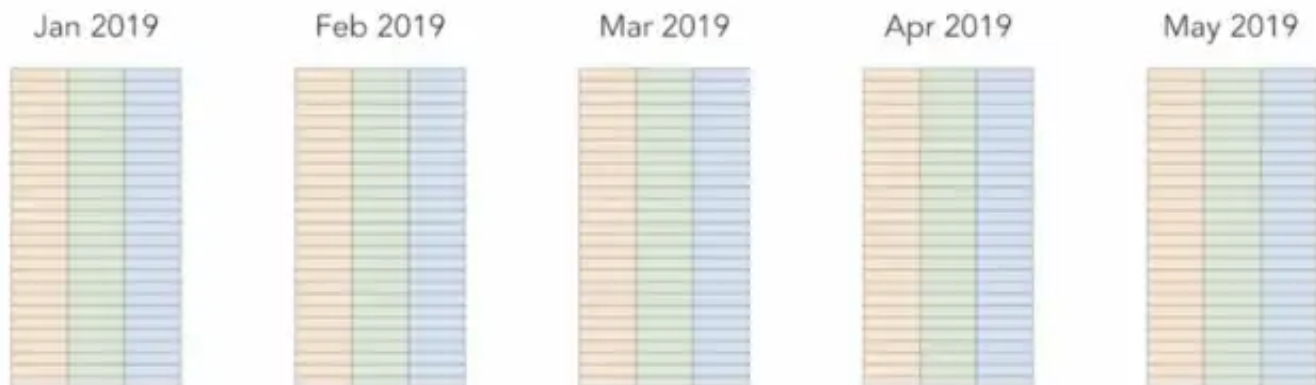


Index

# Partition



# Partition Key

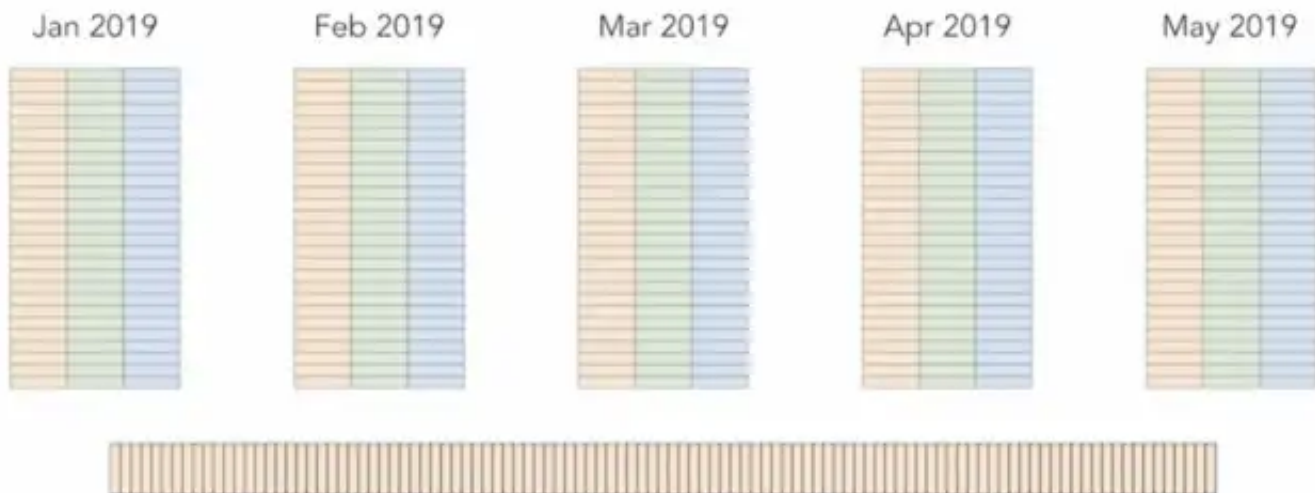




# Local Indexes



# Global Indexes

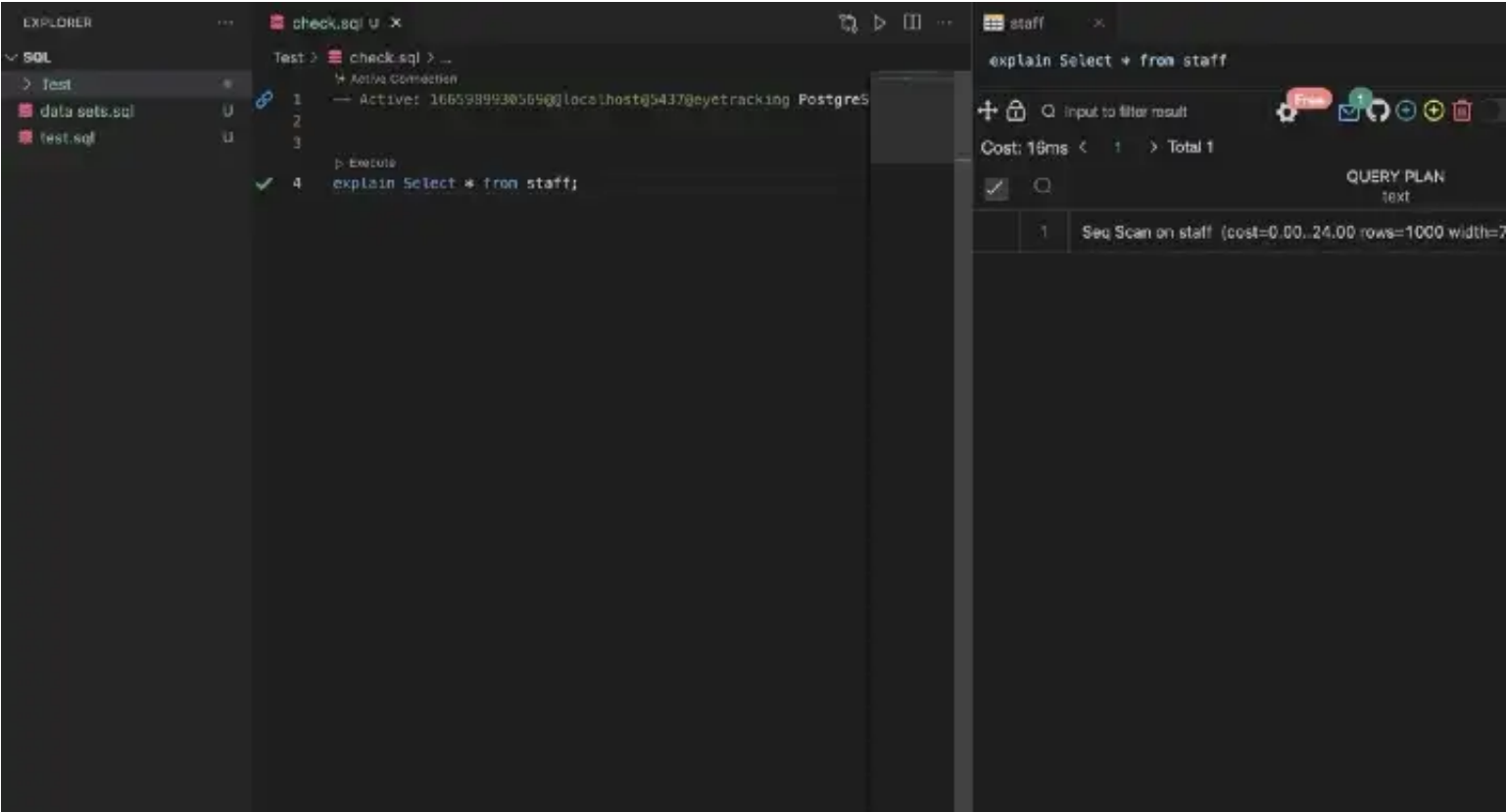


## Explain and analyze

Now, one of the first steps, in optimizing queries, is understanding how they're executing. Remember, SEQUEL is declarative. So, for example, here is a simple declarative query.



Cost is a measure of computation required to complete the step. In this example, the cost spans from 00 to 24.00.



The screenshot shows a PostgreSQL IDE interface. On the left, a sidebar lists database objects including 'Assign 8.0.31', 'assignment 384k', 'eyetracking', 'public', 'tables (13)', 'aois', 'blueprints', 'company\_divis...', 'company\_regio...', 'migrations', 'page\_visit', 'pages', 'participation\_to...', 'projects 9', 'record\_session', 'staff', 'columns', 'id integer', 'last\_name ch...', 'email charact...', 'gender chara...', 'department e...', 'start\_date da...', 'salary integer', 'job\_title char...', 'region\_id inte...', 'index', and 'users'. The main window displays a SQL query editor with the following content:

```
Test > check.sql > ...
Active: 16659899385698@localhost@5437@eyetracking PostgreSQL

1
2
3
4 explain analyze Select email from staff;
```

Below the query editor, the 'QUERY PLAN' section shows the following details:

Step	Operation	Cost	Rows	Width	Actual Time	Rows	Loops
1	Seq Scan on staff	(cost=0.00..24.00)	rows=1000	width=21	(actual time=0.012..0.160)	rows=1000	loops=1
2	Planning Time	0.061 ms					
3	Execution Time	0.233 ms					

If you're working with large data sets, or are concerned about how much data is returned by a query, you can use the row count and the width, to help guide you as you try to reduce the amount of data that's returned.

Where clasuse ( Execution Time / Computation Time )

The screenshot shows a PostgreSQL IDE interface. On the left, a sidebar lists database objects including 'Assign 8.0.31', 'assignment 384k', 'eyetracking', 'public', 'tables (13)', 'aois', 'blueprints', 'company\_divis...', 'company\_regio...', 'migrations', 'page\_visit', 'pages', 'participation\_to...', 'projects 9', 'record\_session', 'staff', 'users', 'users\_projects\_', 'views', 'procedures', and 'functions'. The main window displays a SQL query editor with the following content:

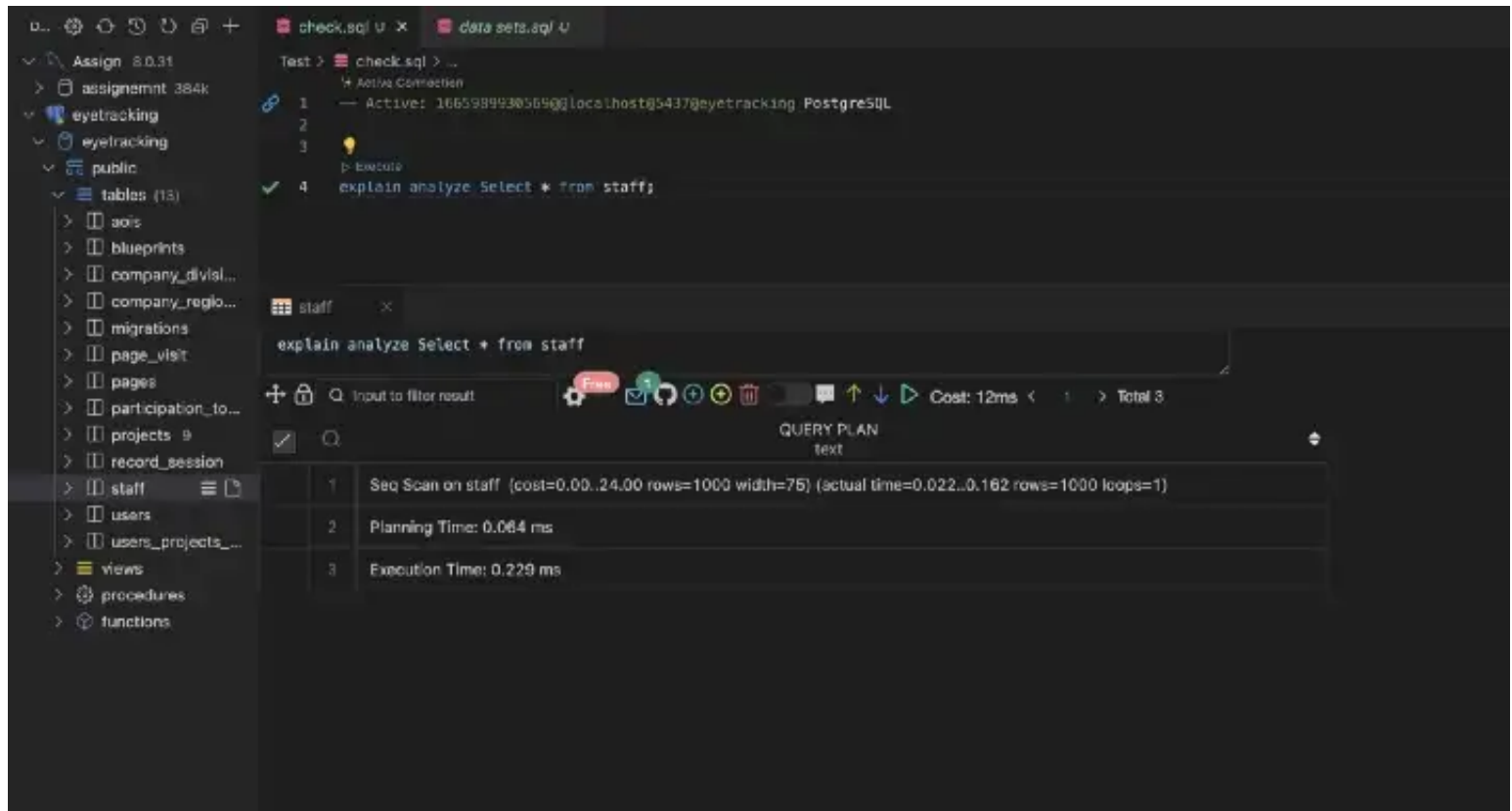
```
Test > check.sql > ...
Active: 16659899385698@localhost@5437@eyetracking PostgreSQL

1
2
3
4 explain analyze Select * from staff
5 where salary > 1000;
```

Below the query editor, the 'QUERY PLAN' section shows the following details:

Step	Operation	Cost	Rows	Width	Actual Time	Rows	Loops
1	Seq Scan on staff	(cost=0.00..26.50)	rows=1000	width=75	(actual time=0.013..0.205)	rows=1000	loops=1
2	Filter: (salary > 1000)						
3	Planning Time	0.297 ms					
4	Execution Time	0.282 ms					

Computation Time = 0.25.50



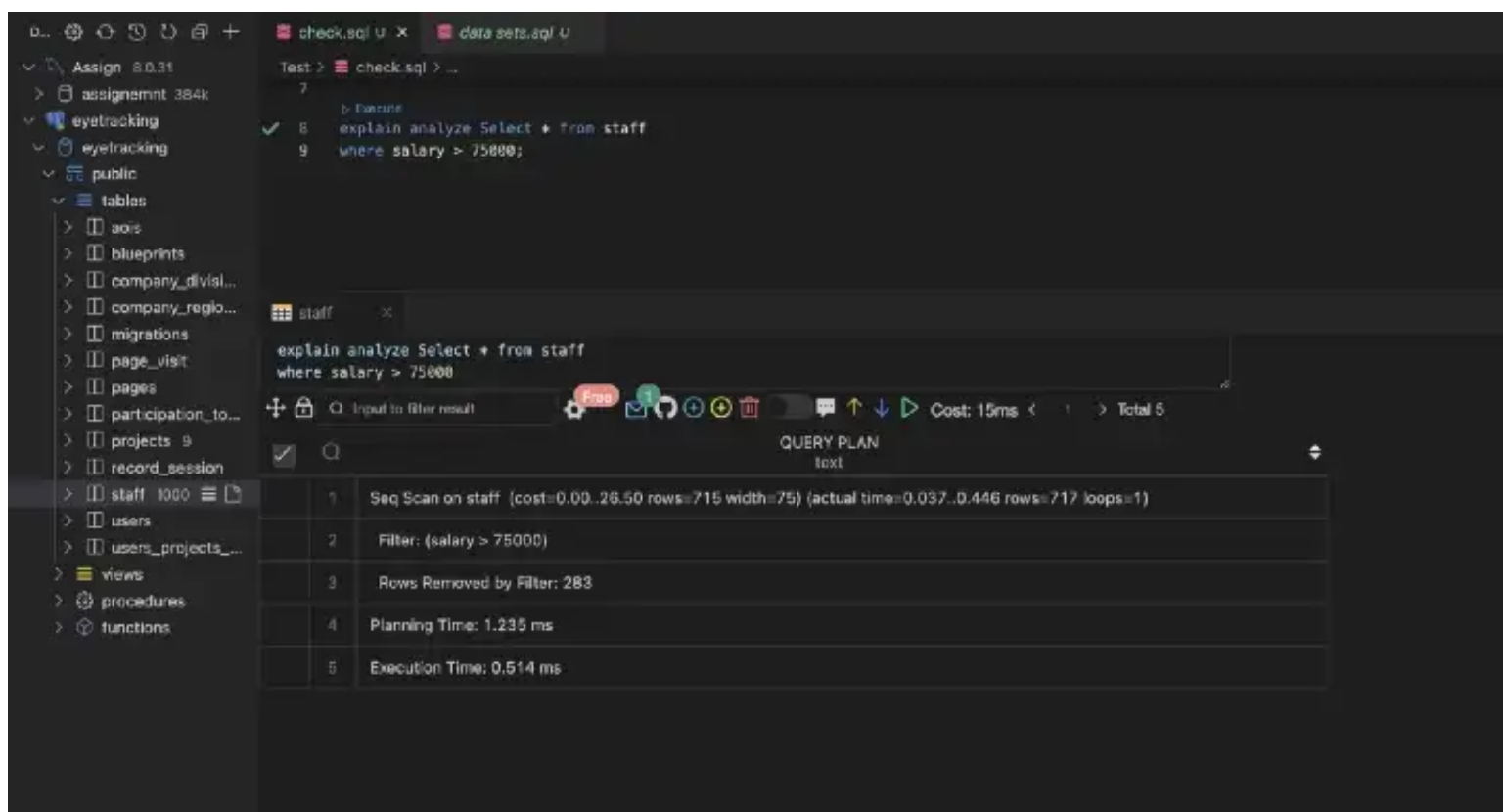
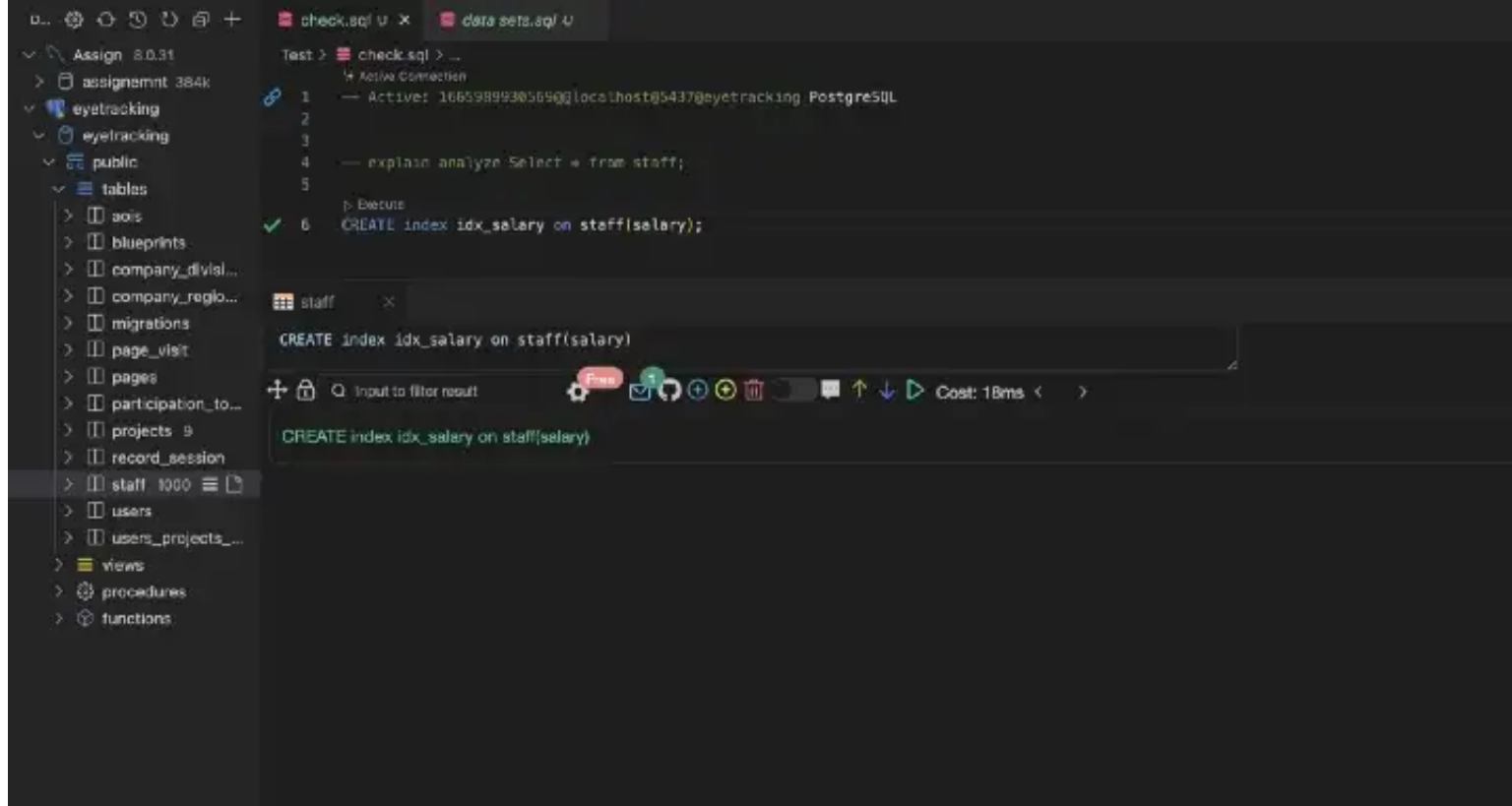
Computation Time = 0.24.00

**Conslusion :** So the total estimated time can actually be less when returning more data if there are fewer steps in the execution plan.

Now come *Indexes*

So the total estimated time can actually be less when returning more data if there are fewer steps in the execution plan.

One of the nice things about indexes is that you can include multiple columns, But for now we are doing Indexing on salary.



The reason is that there are so many rows with a salary greater than 75000, that the query execution builder determined it would actually be faster to simply scan the while table rather than look up those rows in the index, and then actually read the table.

This is a case where our where clause is not selective enough to warrant using an index. So let's try a different salary cut off. Instead of 75000, let's try 150000.

The screenshot shows a database IDE with a sidebar on the left containing a tree view of database objects. The main window displays a SQL query editor with the following text:

```
Test > check.sql > ...  
7  
8 explain analyze Select * from staff  
9 where salary > 150000;
```

Below the query editor, a 'QUERY PLAN' window is open, showing the execution plan for the query. The plan consists of four steps:

Step	Operation
1	Index Scan using idx_salary on staff (cost=0.28..8.29 rows=1 width=75) (actual time=0.009..0.009 rows=0 loops=1)
2	Index Cond: (salary > 150000)
3	Planning Time: 1.301 ms
4	Execution Time: 0.211 ms

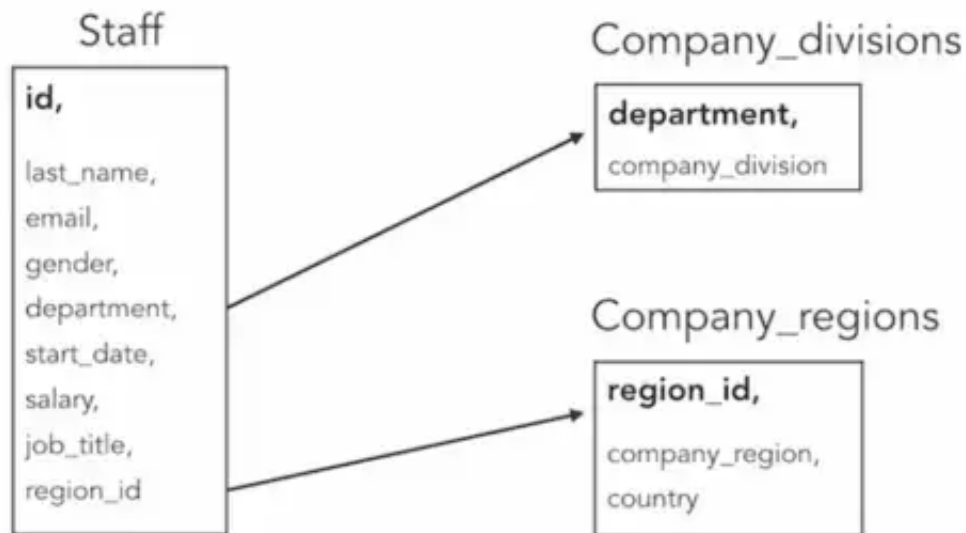
Type of Index

We'll start our learning of indexes by looking at a simple data model.

Our example has three tables.

One with information about employees at a company, we'll call that the staff table. We also have two tables with reference data.

# Example Data Model



Now what is the purpose of indexes

## Purpose of Indexes

- Speed up access to data
- Help enforce constraints
- Indexes are ordered
- Typically smaller than tables



So, Inshort Index reduce scanning :)

The reason that index lookups are faster is that indexes are smaller and they are ordered.

The **big advantage of indexes** is that they reduce the need for **full-table scans**.

Another factor that makes indexes so helpful with querying is that indexes tend to be smaller than their corresponding tables. This means that they're more likely to fit into memory.

That's great news for querying, because reading data from memory is much faster than reading data from a hard disk, or even from solid state drives, or SSDs.

## Implementing Indexes

- Data structure separate from table
- Sometimes duplicates some data, for example, key
- Organized differently than table data



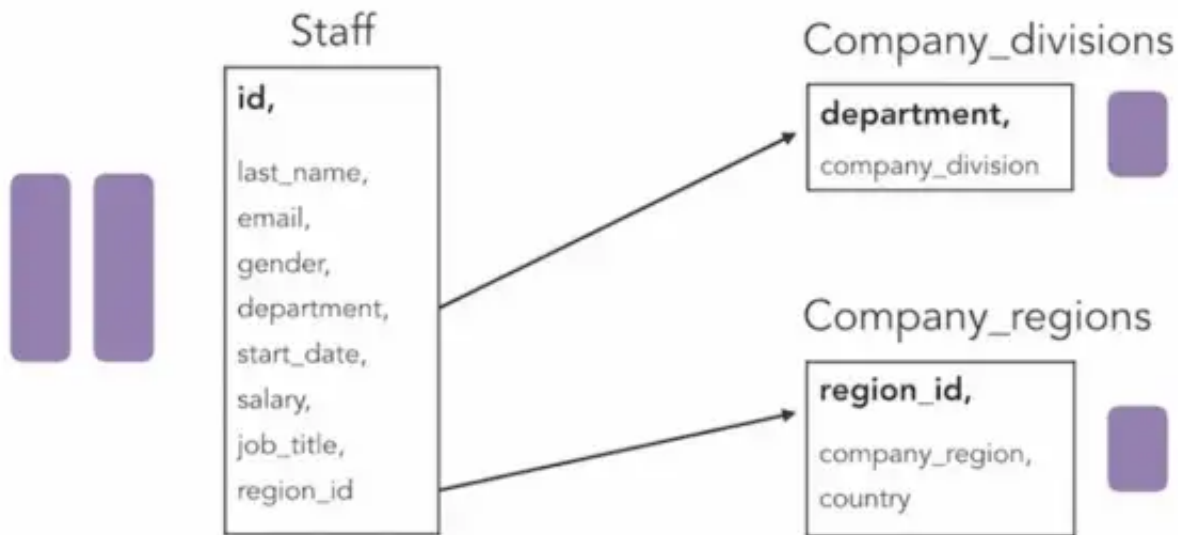
Indexes use storage space in addition to what is in the table. Now usually, we try not to duplicate data. But indexes are a special case.

Their contribution to efficient query processing outweighs the cost of additional storage. Now, this is not always the case.

For example, if most queries on a table require a full table scan, then the index may not be used.



# Additional Data Structure



## Types of Indexes

- B-tree
- Bitmap
- Hash
- Special purpose indexes

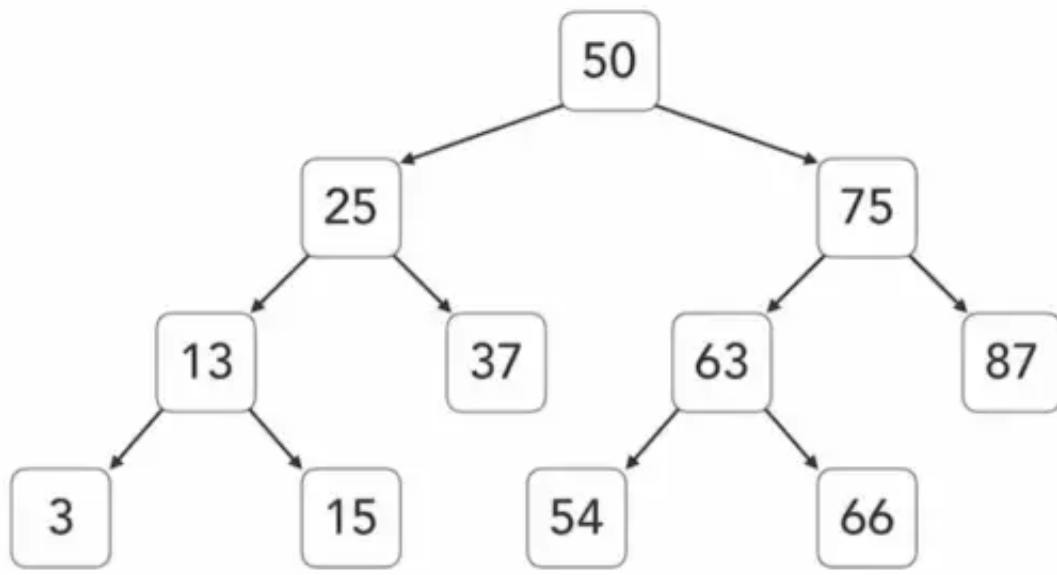


### B-Tree Indexes

The B-tree index is a tree data structure with a root and nodes.

The tree is balanced because the root node is the index value that splits the range of values found in the index column.

# B-Tree Indexes



B-trees are the most commonly used type of index.

It's used when there are a large number of distinct values in a column.

This is called high cardinality.

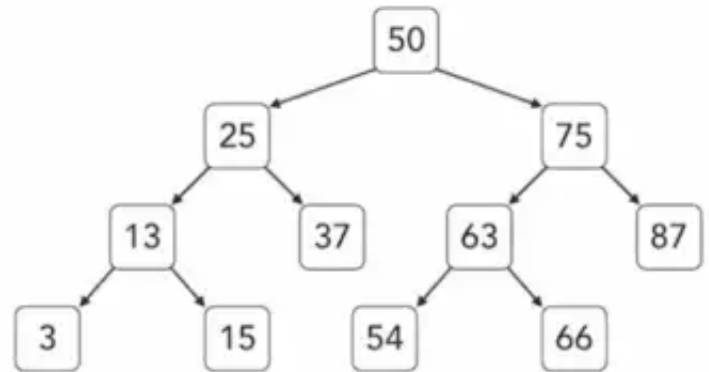
B-trees also rebalance as needed to keep the depth of the tree about the same for all paths.

This prevents a lopsided tree that would be fast to search on one side and slower on the other.

Anytime you look up a value in the B-tree index, you can expect it to take a time that is proportional to the log of the number of nodes in the tree.

# B-Tree Uses

- Most common type of index
- Used when a large number of possible values in a column (high cardinality)
- Rebalances as needed
- Time to access is based on depth of the tree (logarithmic time)



Test > `check.sql` > ...

```
6 -- CREATE INDEX idx_salary ON staff(salary);
7
8 -- explain analyze Select * from staff
9 -- where salary > 150000;
10
11
12 explain analyze Select * from staff
13 where email = 'jellis4@sciencedirect.com';
```

staff

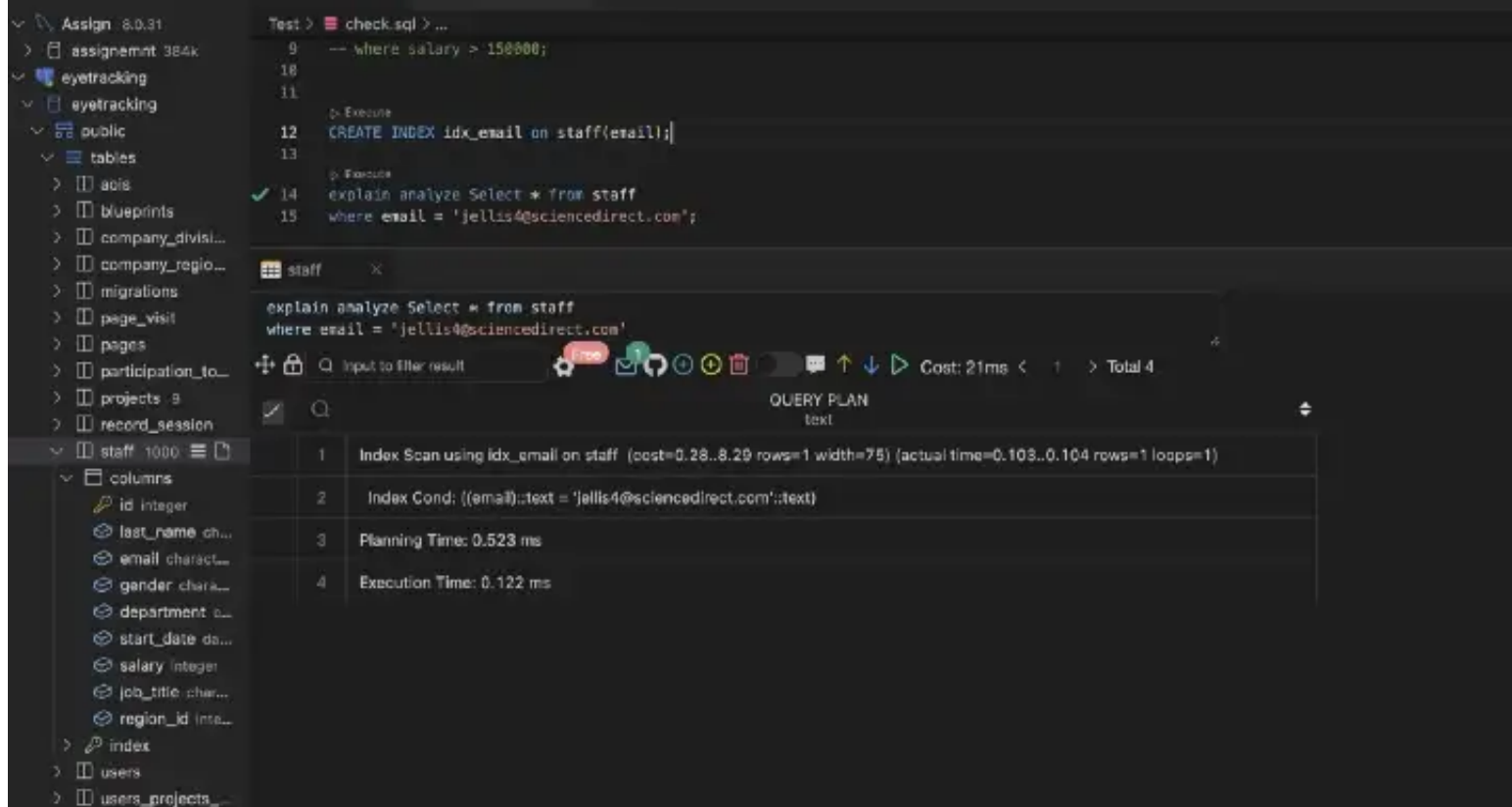
explain analyze Select \* from staff  
where email = 'jellis4@sciencedirect.com'

Cost: 10ms < 1 > Total 5

QUERY PLAN  
text

1	Seq Scan on staff (cost=0.00..26.50 rows=1 width=75) (actual time=0.020..0.160 rows=1 loops=1)
2	Filter: ((email)::text = 'jellis4@sciencedirect.com')::text
3	Rows Removed by Filter: 999
4	Planning Time: 0.090 ms
5	Execution Time: 0.190 ms

Now B tree is a default type of index, so we can use a basic create index command without specifying the B tree specifically.



## BitMap Index

Bitmap Indexes store a series of bits for indexed values. The number of bits used is the same as the number of distinct values in a column.

## Bitmap Uses

- Used when small number of possible values in a column (low cardinality)
- Filter by bitwise operations, such as AND, OR, and NOT
- Time to access is based on time to perform bitwise operations

id	is_union_member	Yes	No
1	Yes	1	0
4	No	0	1
34	No	0	1
14	Yes	1	1
576	Yes	1	1
312	No	0	1
178	NULL	0	0

While Bitmap operations are fast, updating Bitmap Indexes can be more time consuming than other indexes.

They tend to be used in read-intensive use cases, like data warehouses. Some databases, like Oracle, let you create explicit bitmap indexes, but PostgreSQL does not.

## Before

The screenshot shows the PostgreSQL query editor with a query to select staff members with the job title 'Operator'. The query plan shows a sequential scan on the 'staff' table, which is not indexed. The cost is 21ms and the execution time is 0.212 ms.

```
Test > check.sql > ...
10 CREATE INDEX idx_job_title ON staff(job_title);
11
12
13
14
15
16
17
18
19 explain analyze
20 Select +
21 from
22 staff
23 where job_title = 'Operator';
24
```

Cost: 21ms < 1 > Total 5

Step	Operation
1	Seq Scan on staff (cost=0.00..26.50 rows=11 width=75) (actual time=0.021..0.195 rows=11 loops=1)
2	Filter: ((job_title)::text = 'Operator':text)
3	Rows Removed by Filter: 989
4	Planning Time: 0.284 ms
5	Execution Times: 0.212 ms

## After

The screenshot shows the PostgreSQL query editor with the same query as before, but now with an explicit bitmap index created. The query plan shows a bitmap heap scan on the 'staff' table, which is indexed. The cost is 25ms and the execution time is 0.147 ms.

```
15 where email = 'jellis4@sciencedirect.com';
16
17
18 CREATE INDEX idx_job_title ON staff(job_title);
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

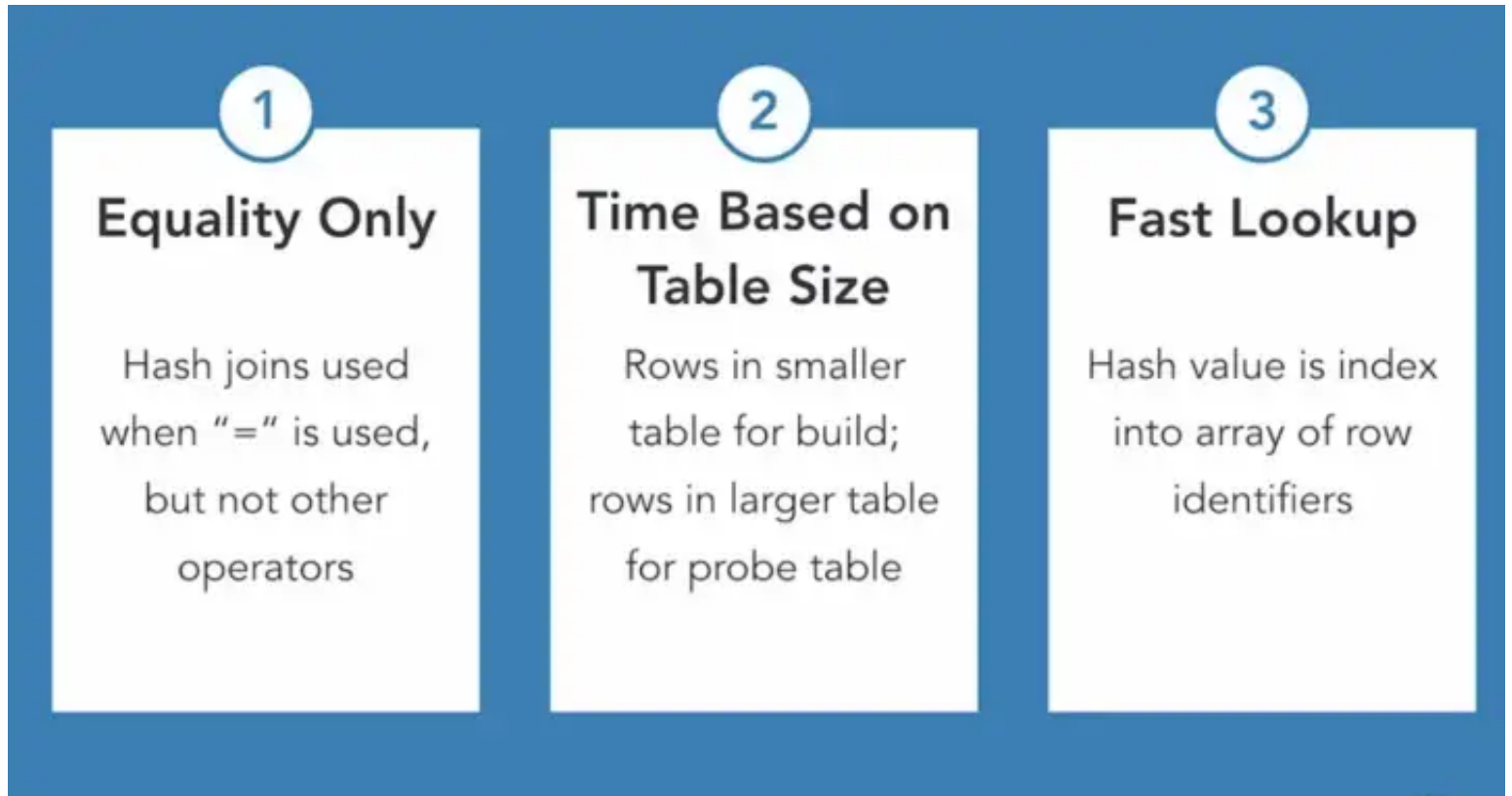
Cost: 25ms < 1 > Total 7

Step	Operation
1	Bitmap Heap Scan on staff (cost=4.36..18.36 rows=11 width=75) (actual time=0.094..0.110 rows=11 loops=1)
2	Recheck Cond: ((job_title)::text = 'Operator':text)
3	Heap Blocks: exact=9
4	-> Bitmap Index Scan on idx_job_title (cost=0.00..4.36 rows=11 width=0) (actual time=0.085..0.085 rows=11 loops=1)
5	Index Cond: ((job_title)::text = 'Operator':text)
6	Planning Time: 0.518 ms
7	Execution Times: 0.147 ms

The query plan uses a bitmap heap scan which only visits data blocks that are needed and does not scan all index blocks.

Now bitmap indexes are created on the fly by Postgres when it thinks it will be useful.

## Hash Join



Hash joins use a function to map data into a value that can act as an index into an array.

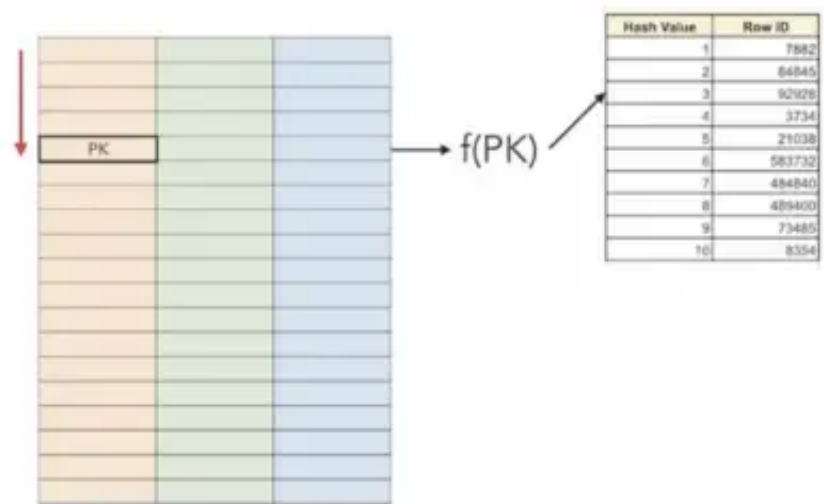
Hash functions are designed so that the output value is virtually unique across all of the values in the domain, where that is the set of possible input values.

Even slight changes to the input value can lead to different outputs.

Part of the process of hash joining is building a table of hash values.

# Probe Hash Table

- Step through large table
- Compute hash value of primary or foreign key value
- Lookup corresponding value in hash table



The smaller of the two tables is used. For each key value in the smaller table, we compute a hash value and store that in the table.

```
Test > check.sql > ...
p: Execute
28 explain analyze SELECT
29   s.id, s.last_name
30 from staff s
31 JOIN
32   company_regions
33 using (region_id)
34

staff
explain analyze SELECT
  s.id, s.last_name
from staff s
JOIN
  company_regions
using (region_id)

QUERY PLAN
text
1 Hash Join (cost=22.38..49.02 rows=1000 width=11) (actual time=0.091..0.510 rows=1000 loops=1)
2 Hash Cond: (s.region_id = company_regions.region_id)
3 -> Seq Scan on staff s (cost=0.00..24.00 rows=1000 width=15) (actual time=0.029..0.175 rows=1000 loops=1)
4 -> Hash (cost=15.50..15.50 rows=550 width=4) (actual time=0.030..0.030 rows=7 loops=1)
5 Buckets: 1024 Batches: 1 Memory Usage: 9kB
6 -> Seq Scan on company_regions (cost=0.00..15.50 rows=550 width=4) (actual time=0.015..0.017 rows=7 loops=1)
7 Planning Time: 0.405 ms
8 Execution Time: 0.636 ms
```

## Merge Join

Merge joins are especially useful when tables don't fit into memory, because we can do fewer reads from disks than we would likely have to do if we used a nested loop join.



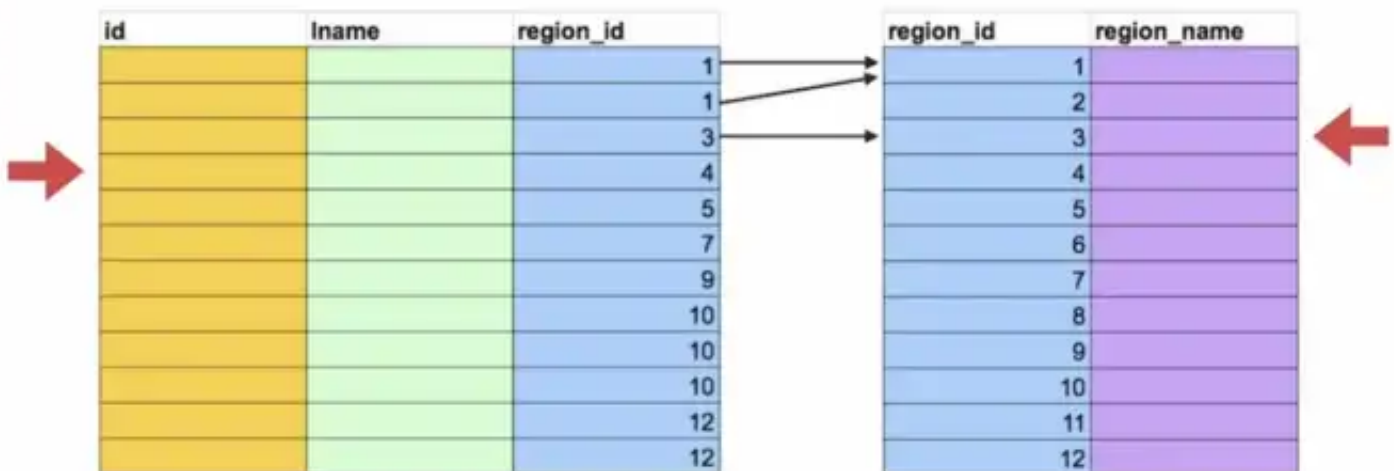
Merge joins are used only with equality join operations

## Sorting

- Merge join also known as sort merge
- First step is sorting tables
- Takes advantage of ordering to reduce the number of rows checked



## Merging



1

## Equality Only

Merge joins used  
when "=" is used  
but not other  
operators

2

## Time Based on Table Size

Time to sort and  
time to scan

3

## Large table joins

Works well when  
tables do not fit in  
memory

Now, note the cost of this mergejoin is about 114/115 computational units. The hashjoin took about 22/24 computational units.

```

28 explain analyze SELECT
29   s.id, s.last_name
30 from staff s
31 JOIN
32   company_regions
33   using (region_id);
34
35
36 p: Execution
37 set enable_nestloop=false;
38 p: Execution
39 set enable_hashjoin=false;
40 p: Execution
41 set enable_mergejoin=true;
42
43

```

staff

```

explain analyze SELECT
  s.id, s.last_name
from staff s
JOIN
  company_regions
  using (region_id)

```

Cost: 7ms < 1 > Total 12

QUERY PLAN

text

1	Merge Join (cost=114.36..132.11 rows=1000 width=11) (actual time=0.858..1.162 rows=1000 loops=1)
2	Merge Cond: (s.region_id = company_regions.region_id)
3	-> Sort (cost=73.83..76.33 rows=1000 width=15) (actual time=0.818..0.883 rows=1000 loops=1)
4	Sort Key: s.region_id

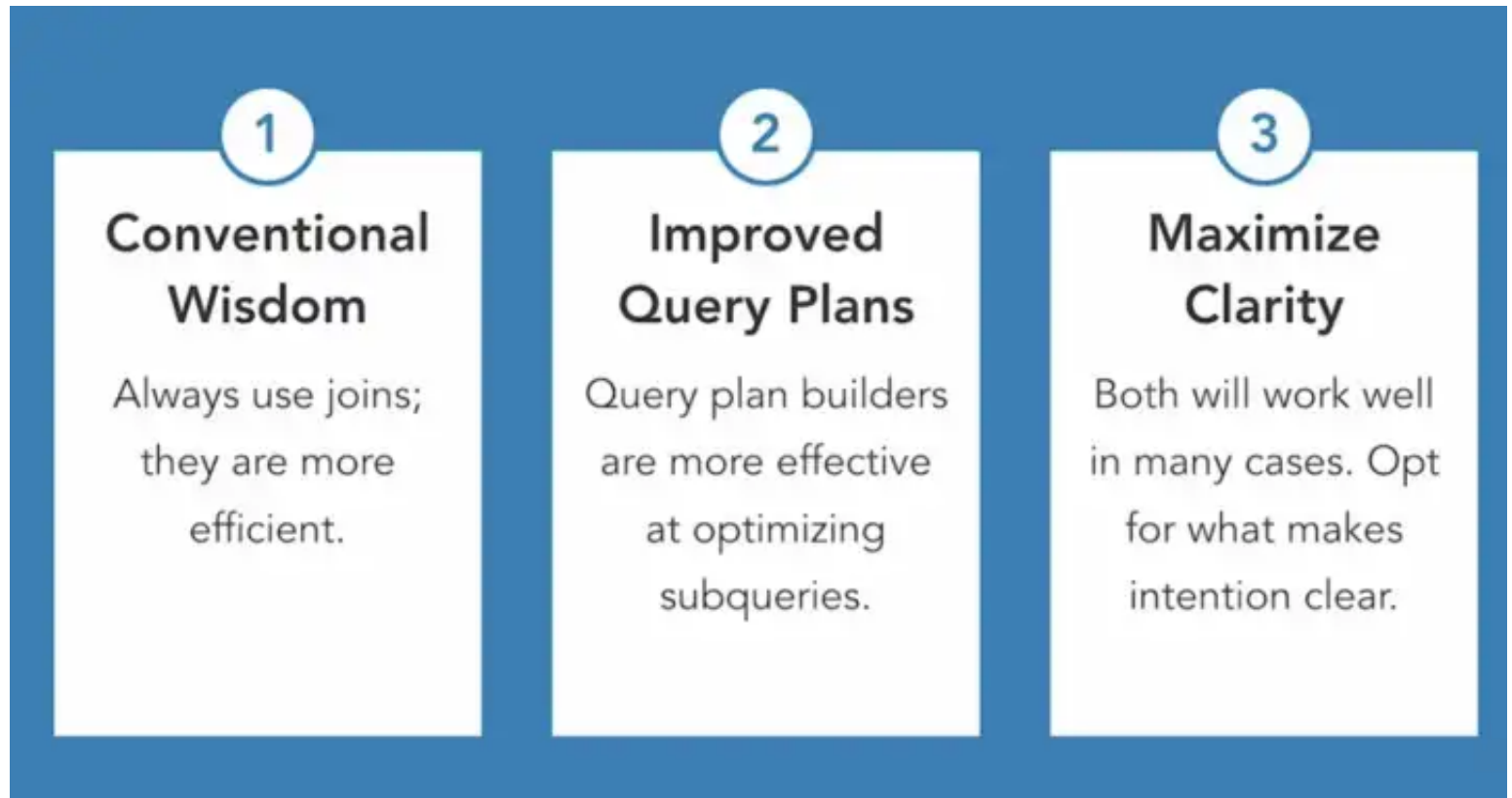
## SubQueries Vs Join

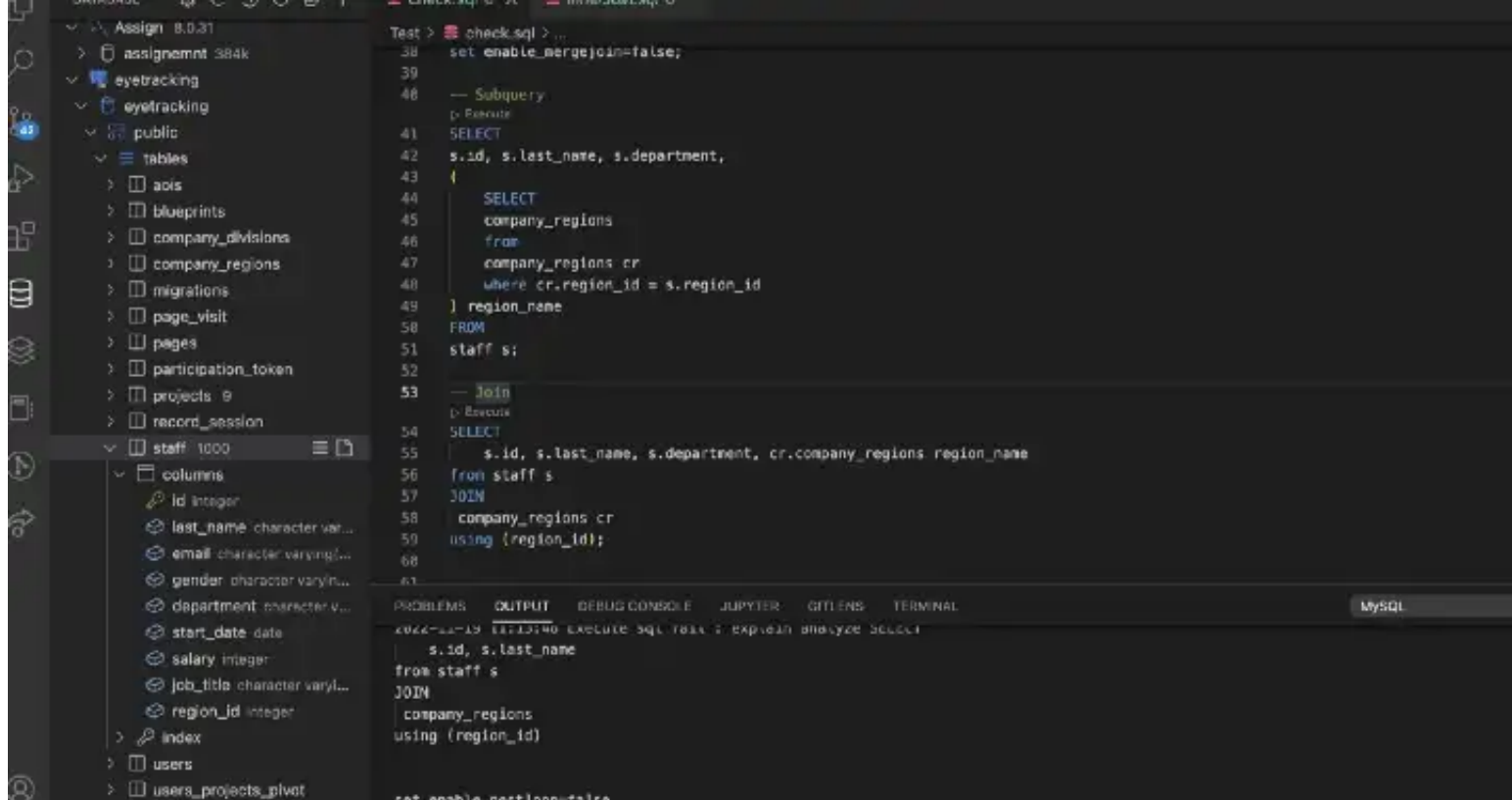
Well if we go by the conventional wisdom, then we should always use joins since they're more efficient.

Now that may have been true in the past, but optimizers have improved over time and can build efficient execution plans for subqueries.

My advice is to optimize for clarity. So use the method that makes your intentions clear.

If there is a significance performance difference between joins and subqueries, then choose the optimal one, and document your query so others, who are reading your code, can immediately understand your intentions.





More on Partitioning Data, Materilized View and Optimization Techinque in next post. Thanks for reading!

Database      Sql      Indexing      Innerjoin      Sql Server