

***> - what does "Bitmap Heap Scan" phase do?***

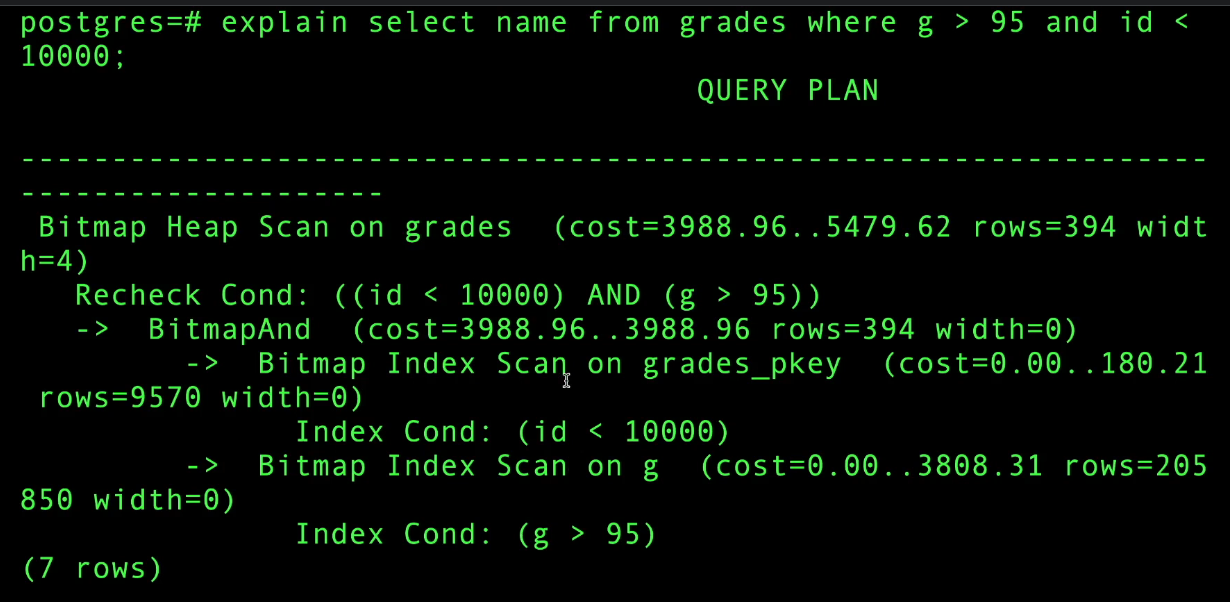
A plain indexscan fetches one tuple-pointer at a time from the index, and immediately visits that tuple in the table. A bitmap scan fetches all the tuple-pointers from the index in one go, sorts them using an in-memory "bitmap" data structure, and then visits the table tuples in physical tuple-location order. The bitmap scan improves locality of reference to the table at the cost of more bookkeeping overhead to manage the "bitmap" data structure --- and at the cost that the data is no longer retrieved in index order, which doesn't matter for your query but would matter if you said ORDER BY.

***> - what is "Recheck condition" and why is it needed?***

If the bitmap gets too large we convert it to "lossy" style, in which we only remember which pages contain matching tuples instead of remembering each tuple individually. When that happens, the table-visiting phase has to examine each tuple on the page and recheck the scan condition to see which tuples to return.

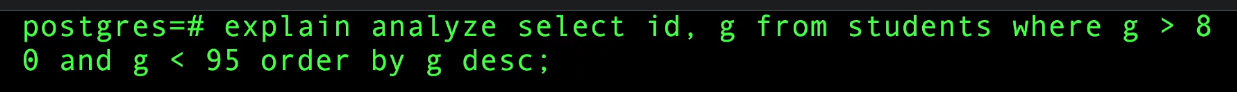
***> - I thought "Bitmap Index Scan" was only used when there are two or more applicable indexes in the plan, so I don't understand why is it used now?***

True, we can combine multiple bitmaps via AND/OR operations to merge results from multiple indexes before visiting the table ... but it's still potentially worthwhile even for one index. A rule of thumb is that plain indexscan wins for fetching a small number of tuples, bitmap scan wins for a somewhat larger number of tuples, and seqscan wins if you're fetching a large percentage of the whole table.

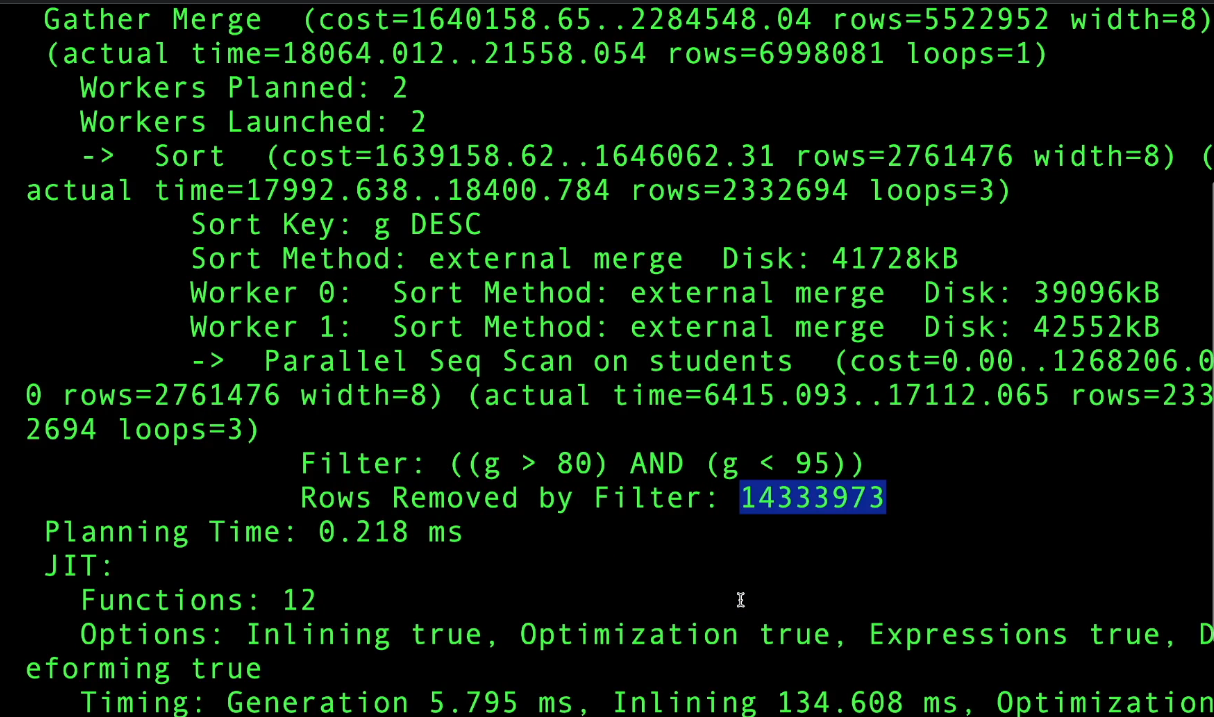


combining both bitmaps with BitmapAnd reducing the number of rows to 394 and then fetch the table

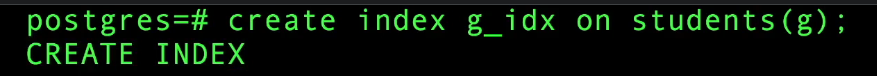
***Key vs non-key column indexes:***

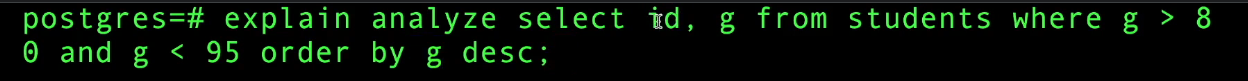


g is not indexed, so this takes long time to execute.

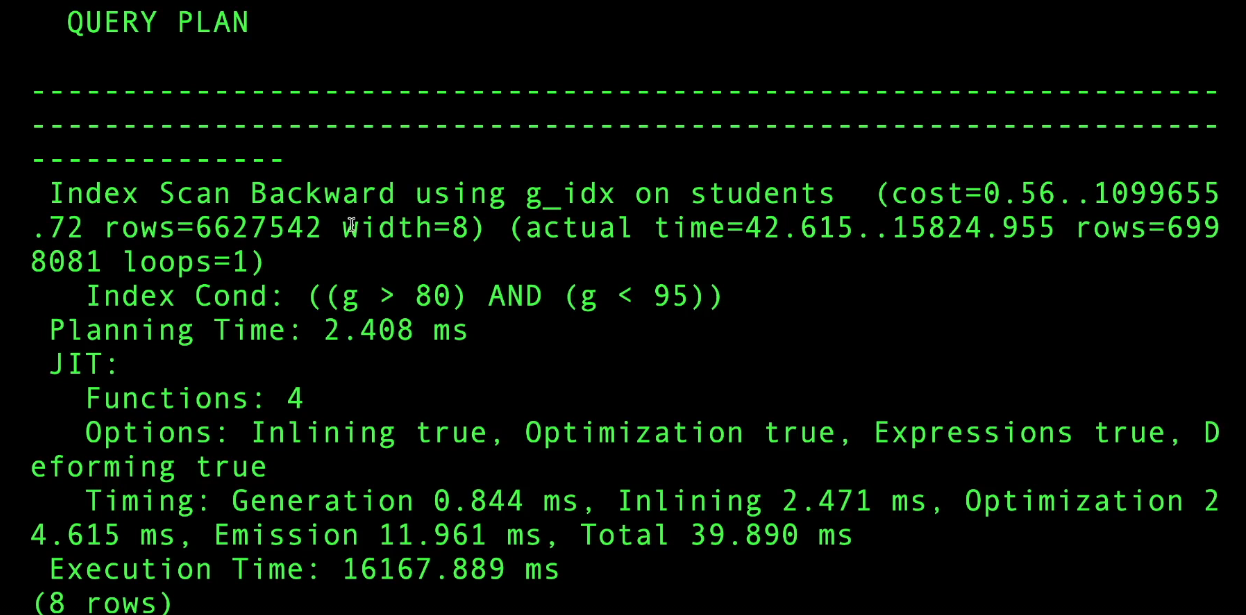


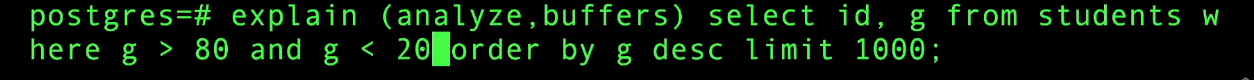
lets create index on g



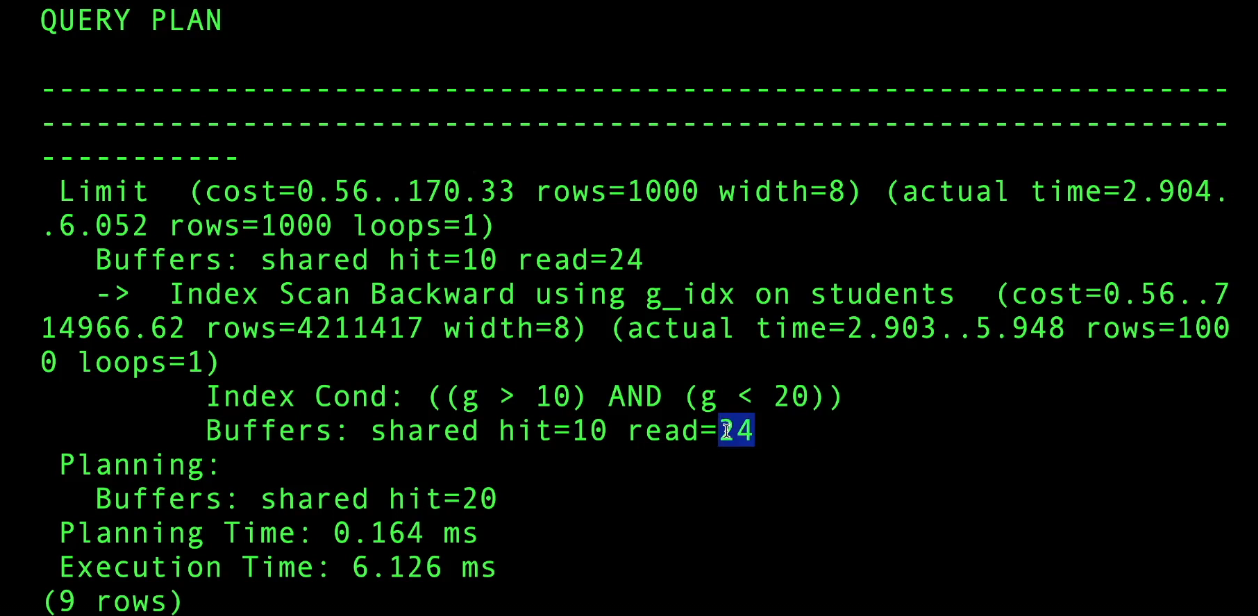


it took 16 secs to execute because it has to go to disk to fetch the id field we are asking for

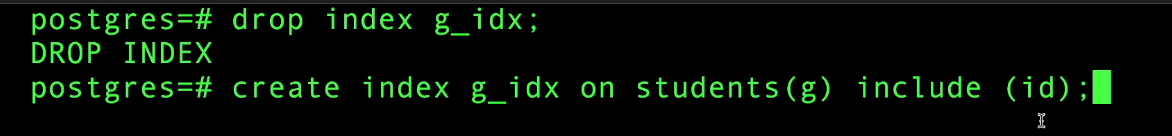


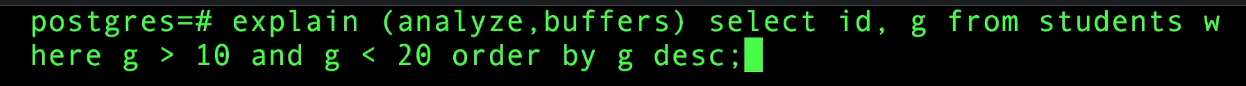


if we run the query again it will be fast because the database caches the previous results

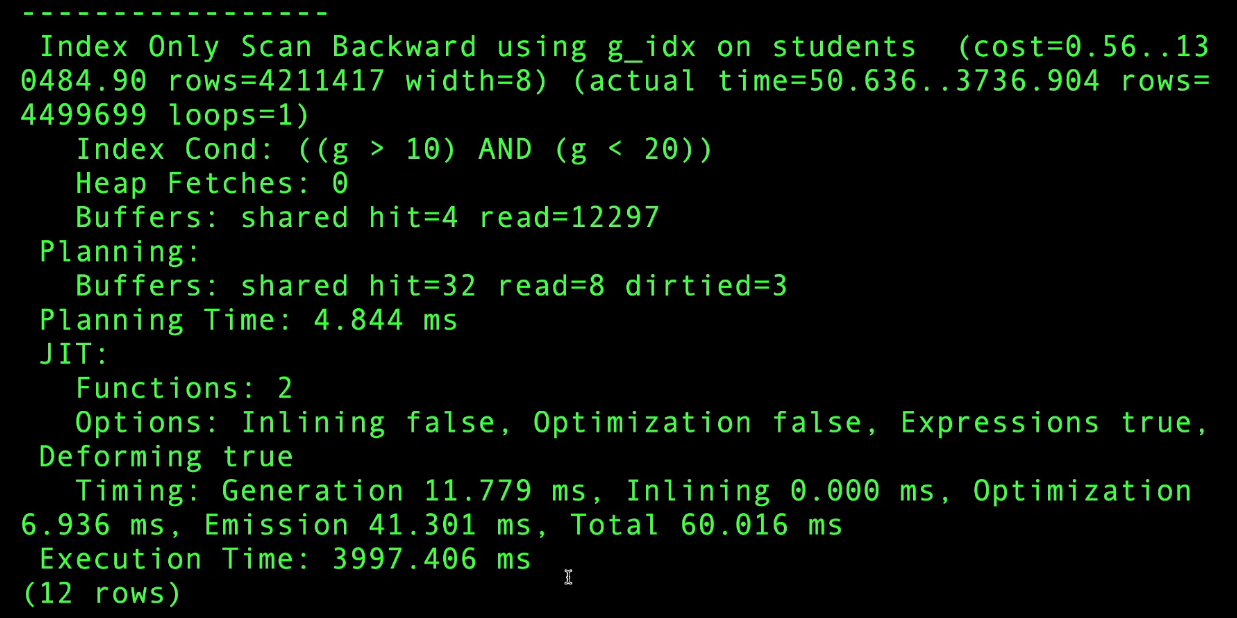


most of the cost is due to io time to fetch id, so lets create a non key index



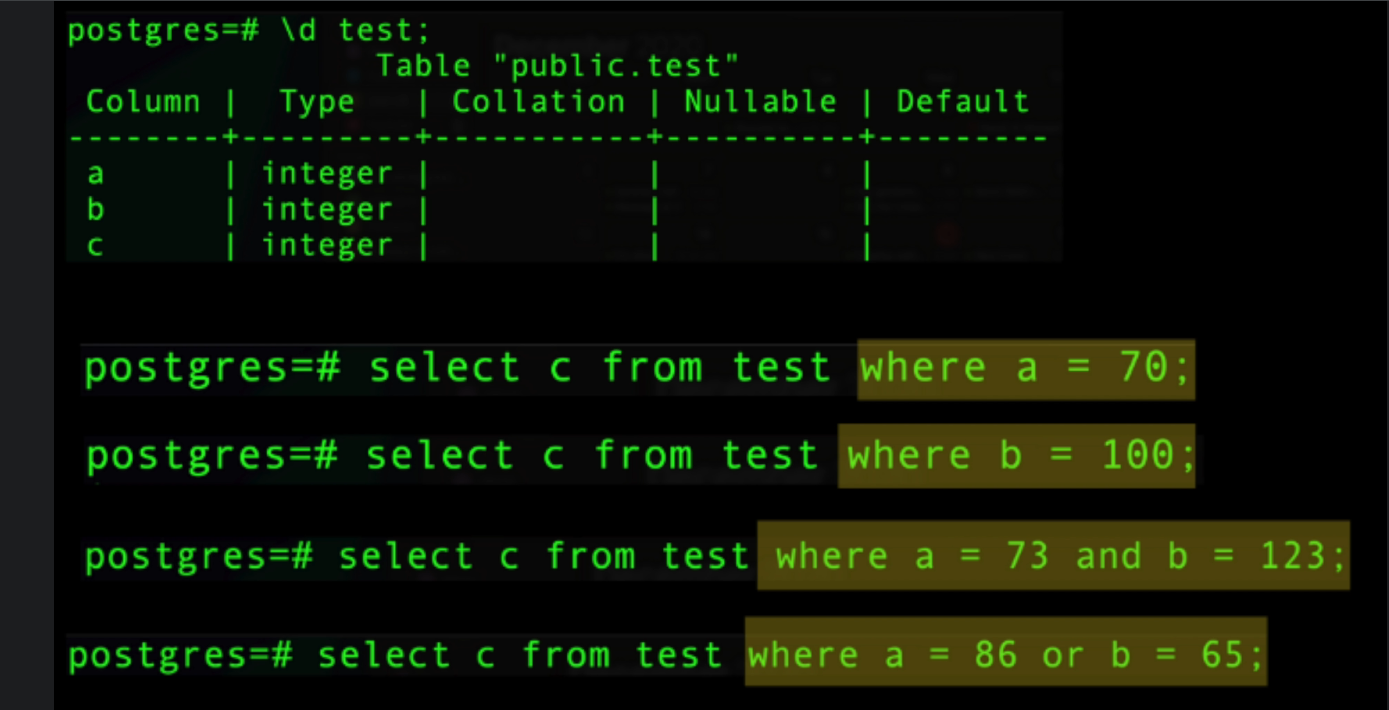


now this is index only scan, it took 4 sconds now, observe that we are still going to io (read = 12297 times) cuz index data structure is stored on disc because its large for 50 mil rows.



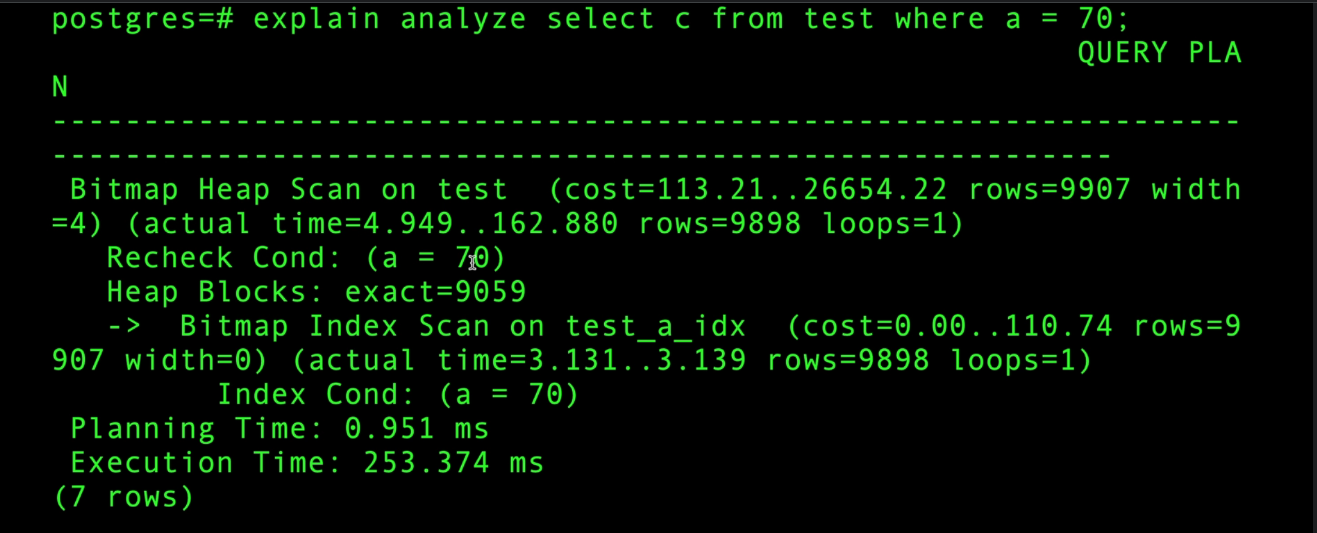
***Combining Database Indexes for better performance***:

Imagine your table structure is like this and you often need to run queries like these.

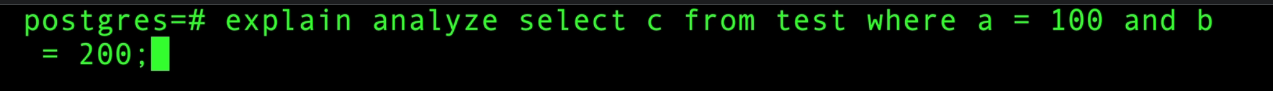




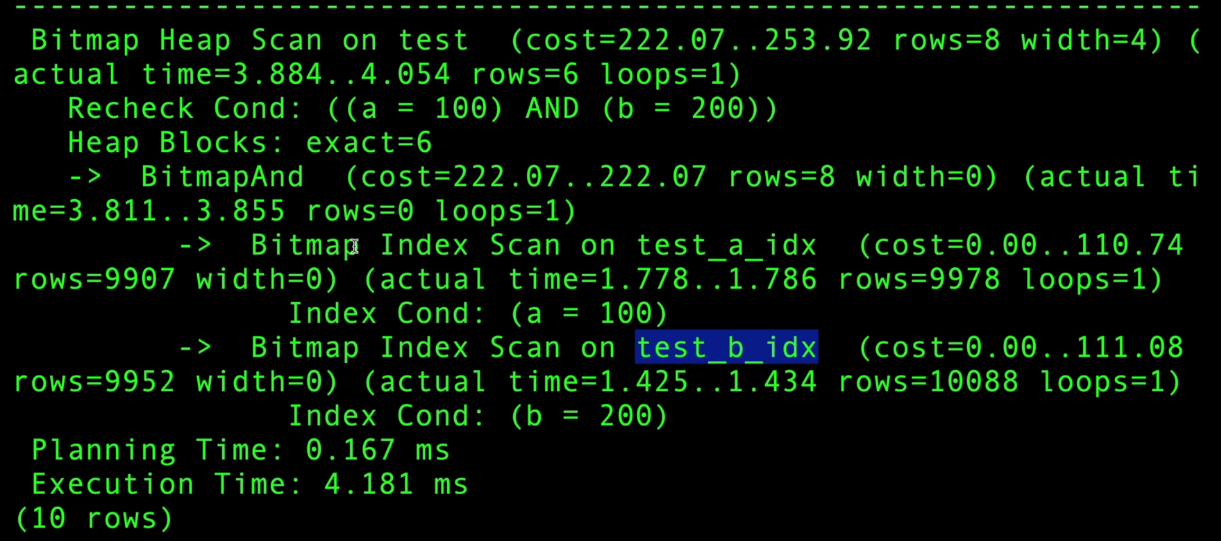
when we have one index on a and one index on b

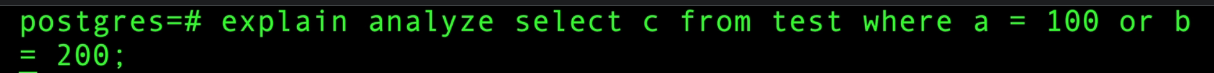


since we might have lots of rows to fetch from disk pg uses bitmap index scan to create bit map and do Bitmap heap scan for better performance.

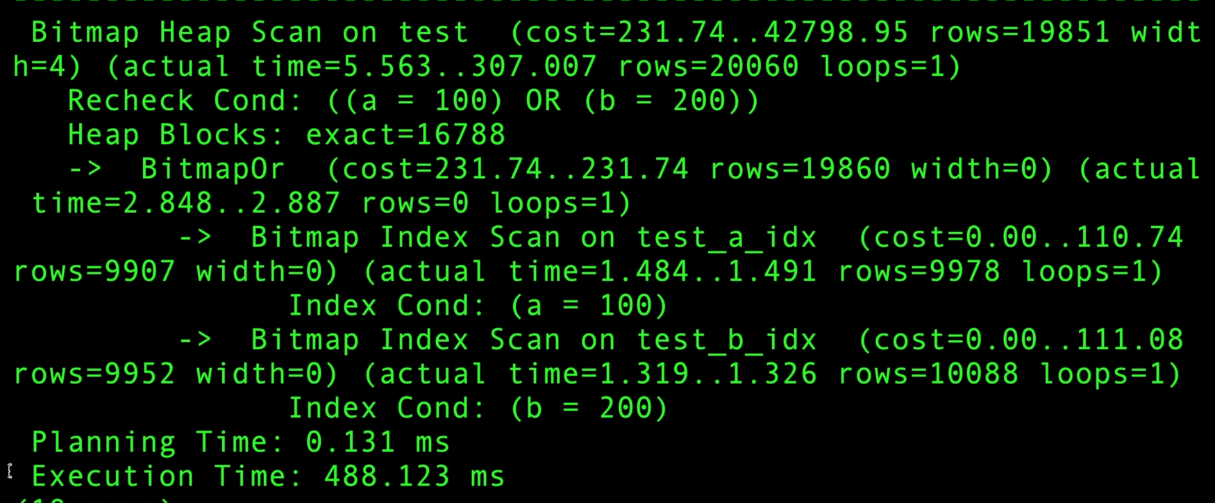


when our filter includes both a and b, pg does parallel bitmap index scan on a and b, then BitmapAnd to combine the bitmaps, then pull the values from the disc and applies recheck condition.

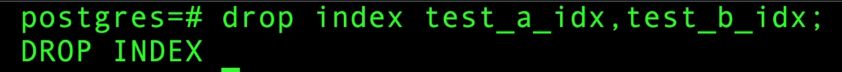




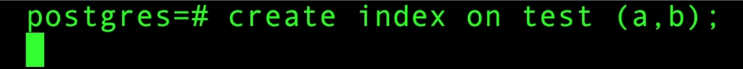
this is for or condition, only difference is BitmapOr, note the execution time is more cuz OR always fetches more rows compared to AND.

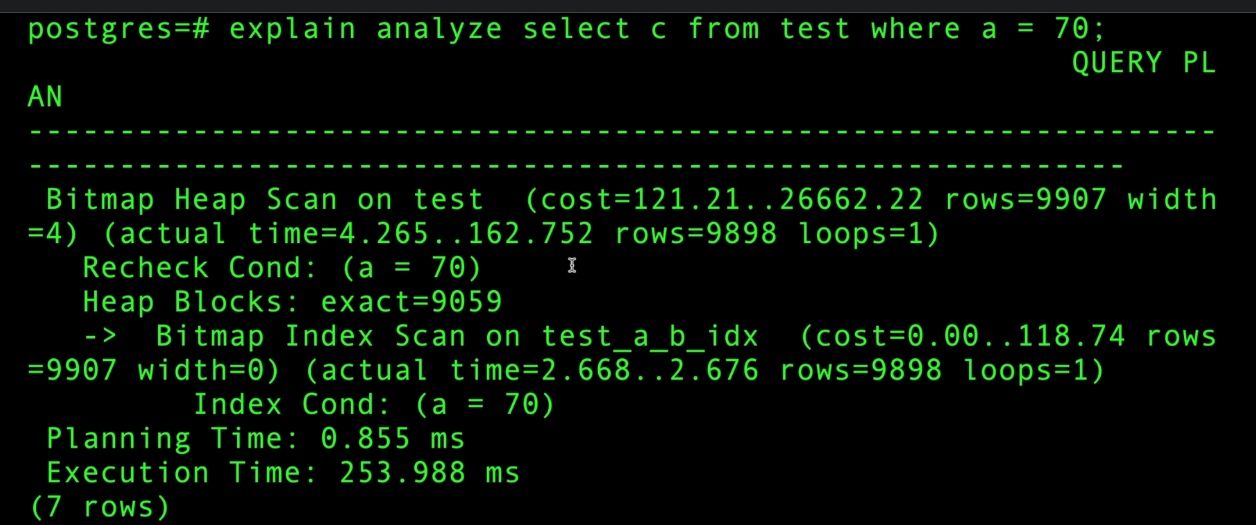


lets drop these indexes and lets create composite index

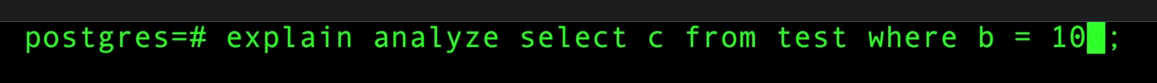


now this composite index is more efficient for queries that filter with both a and b

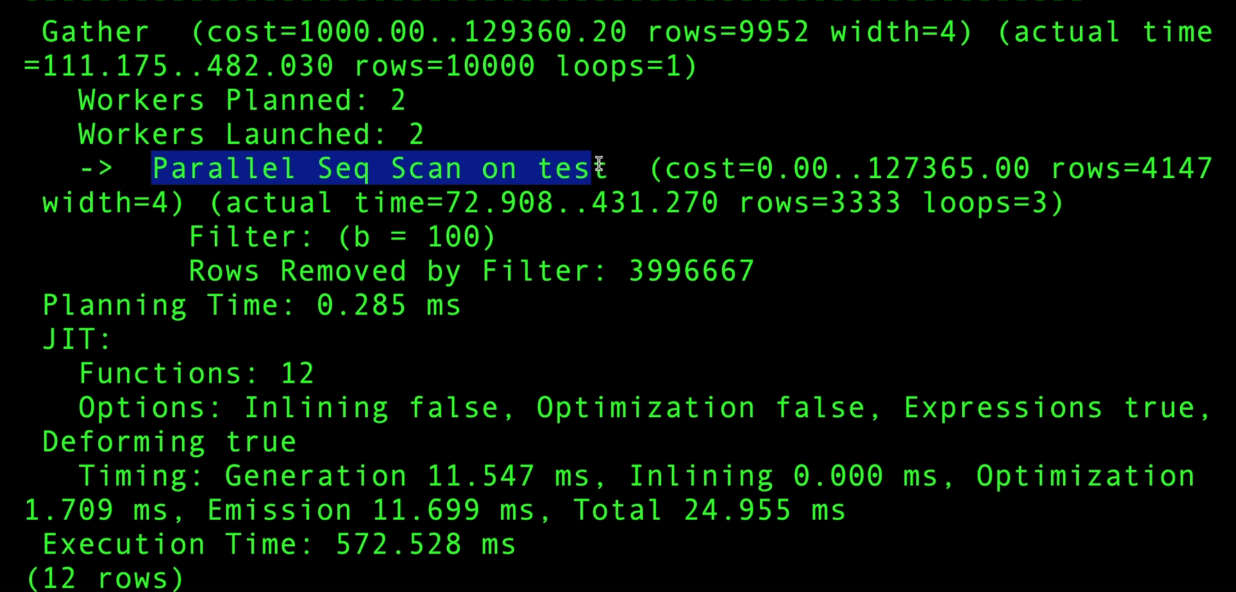




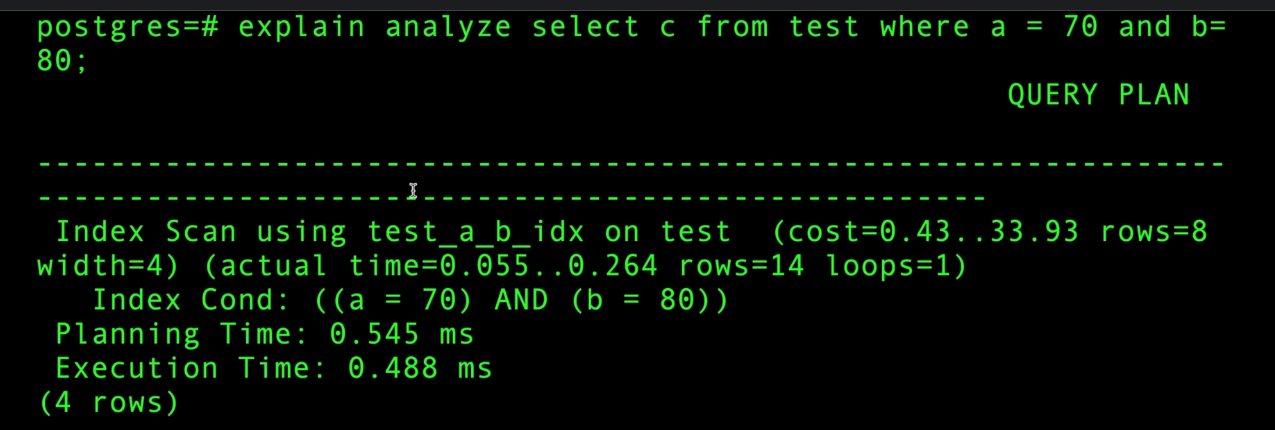
pg decides to use composite index for only a filter, because a is on the left hand side of the created composite index, this is a pg rule.



we can not use the composite index for only b filter, so pg uses seq scan



a and b filter query will be very efficient and uses composite index



creating index blocks all other operations except select so we can use concurrently to avoid this