Pushpak Sharma Following

Nov 19, 2022 · 8 min read · Listen

SQL Query Performance Optimization

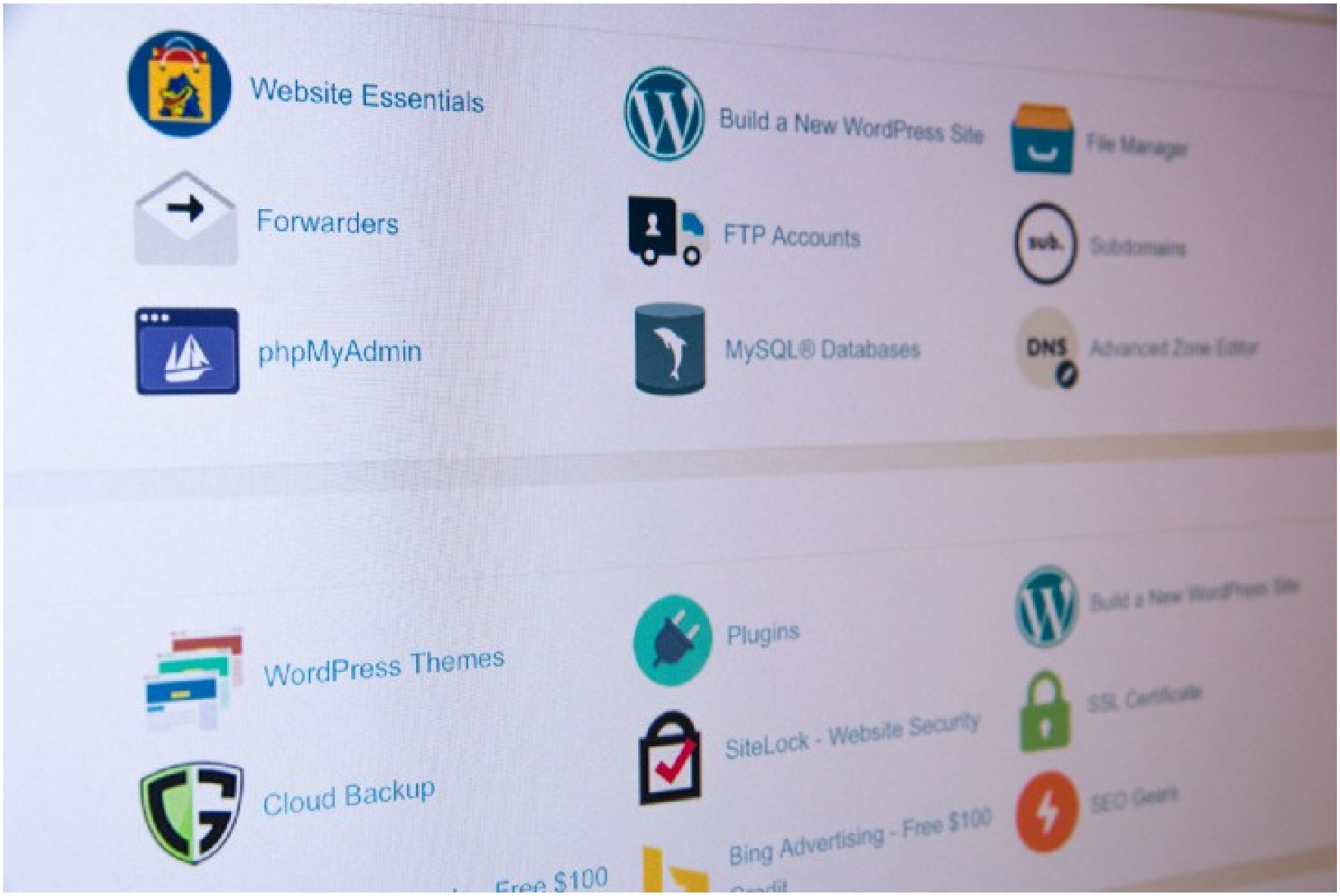


Photo by Stephen Phillips - Hostreviews.co.uk on Unsplash

Objective

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

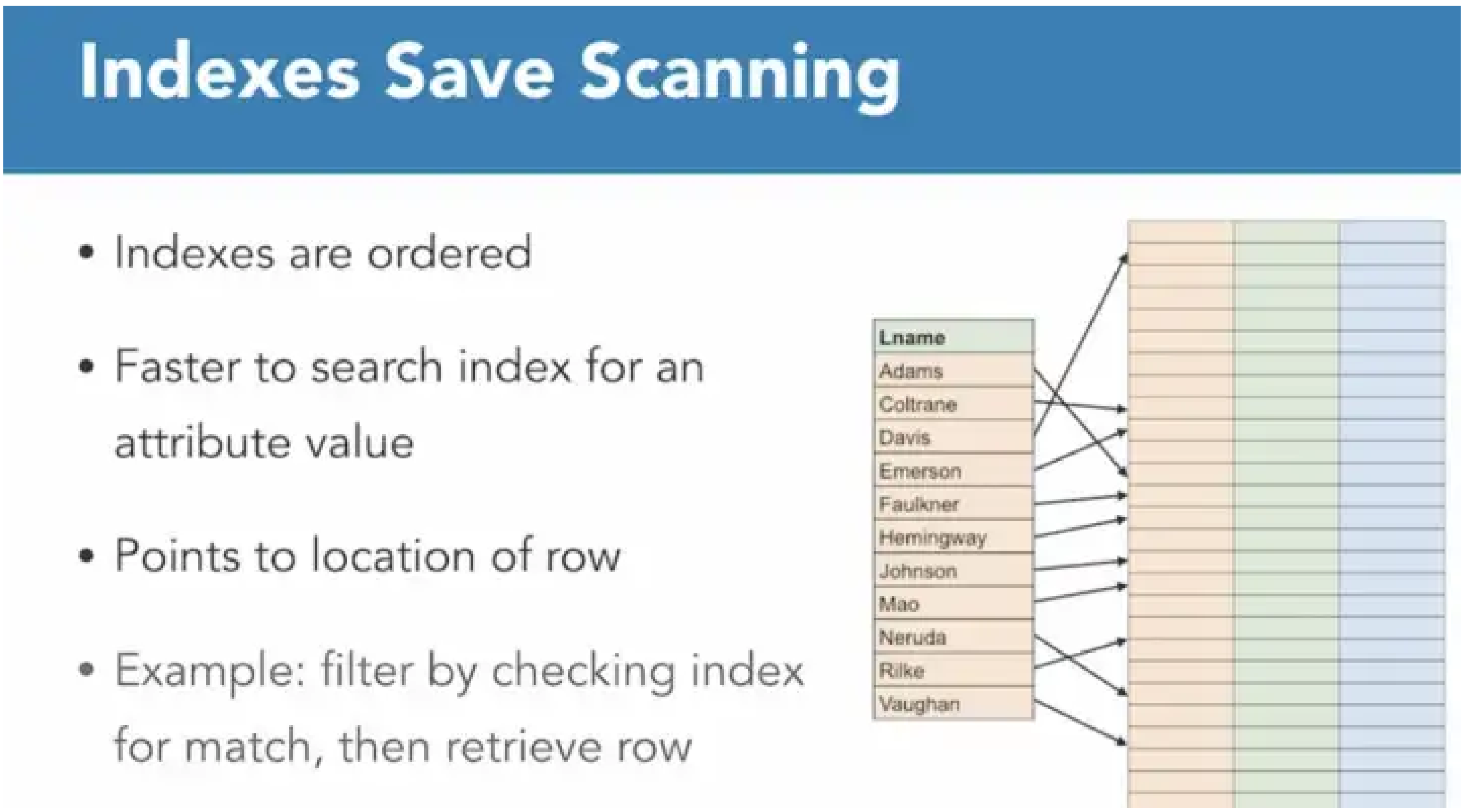
- When to use partitioning to improve performance
- Collecting statistics about data in tables

What is scanning?

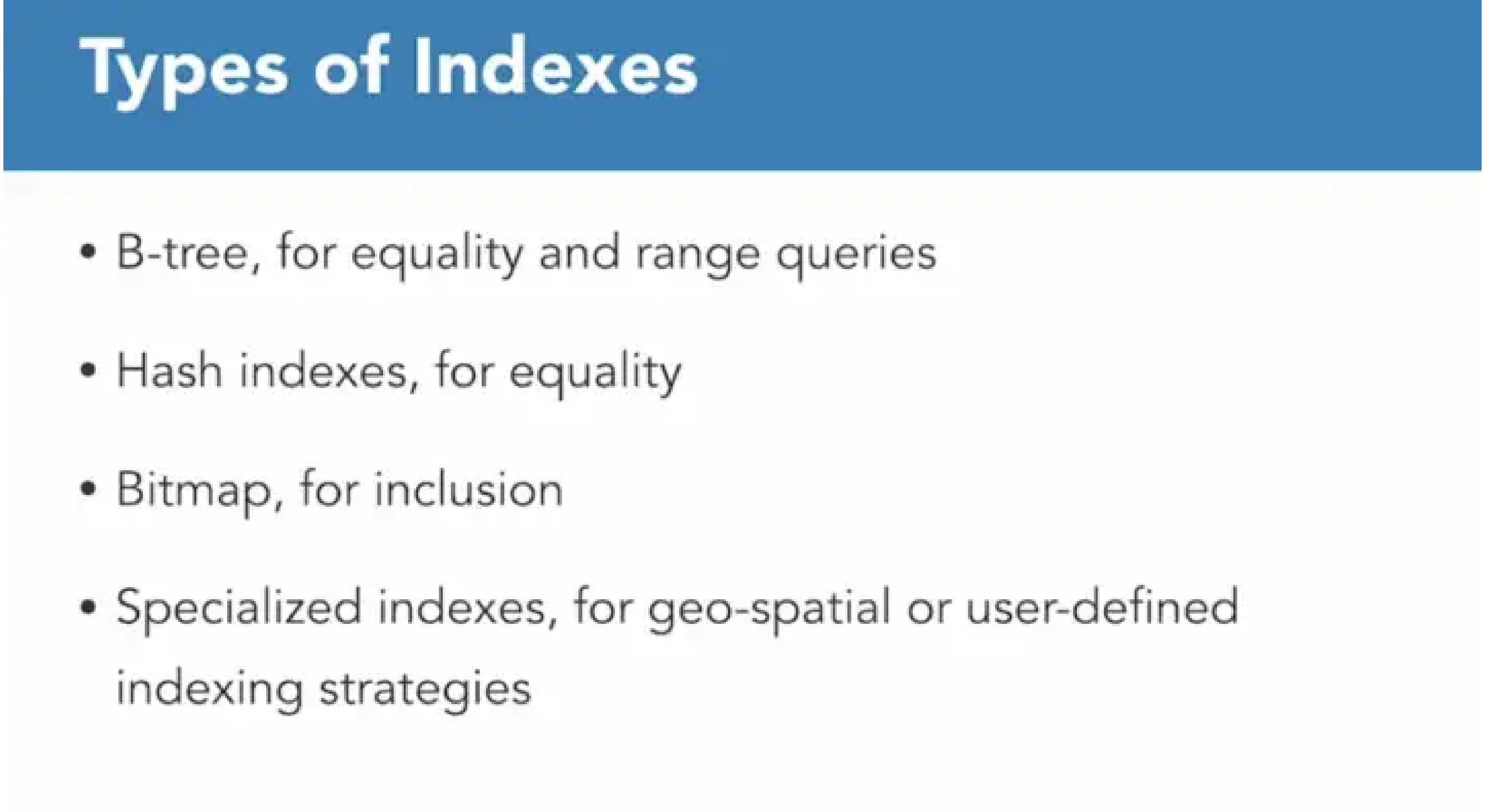
Scanning is a **linear operation**, moving from one row to the next, and performing some operation, like applying a filter to implement a WHERE clause. Scanning is simple.

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.

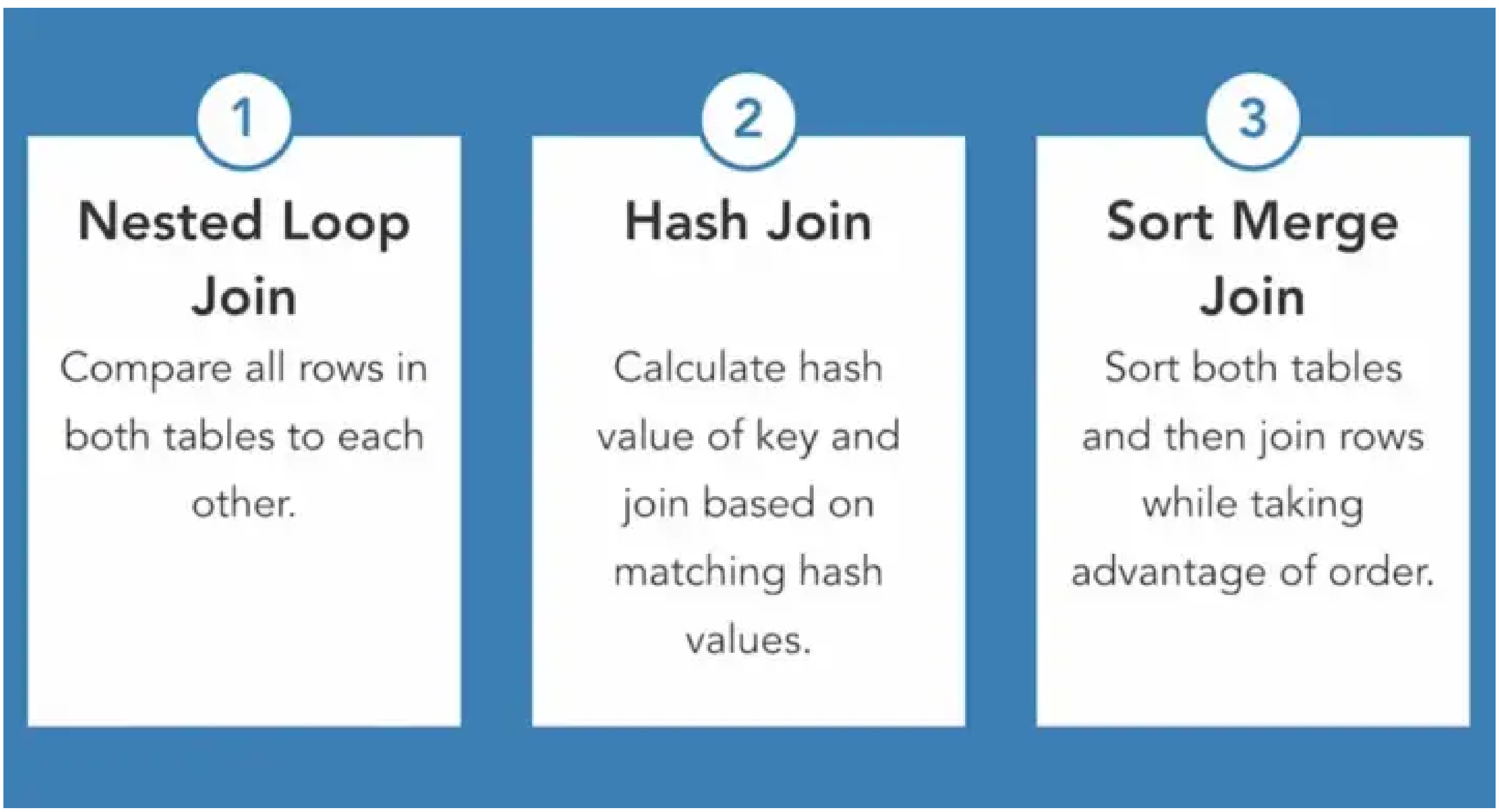
How do Indexes save scanning?



Types of Indexes



Types of Join

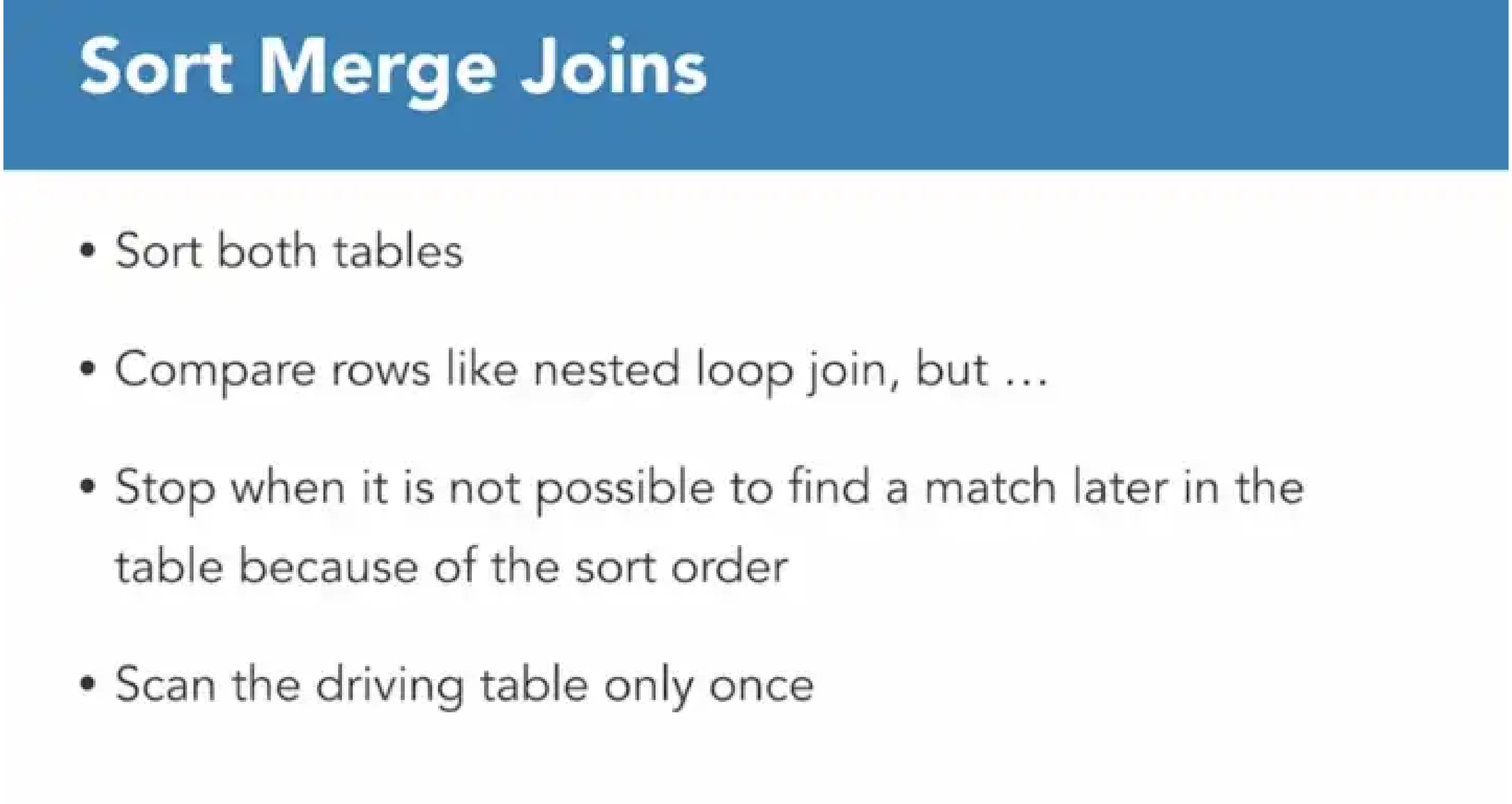


Nested Loop Joins

- 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



What is Partitioning

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

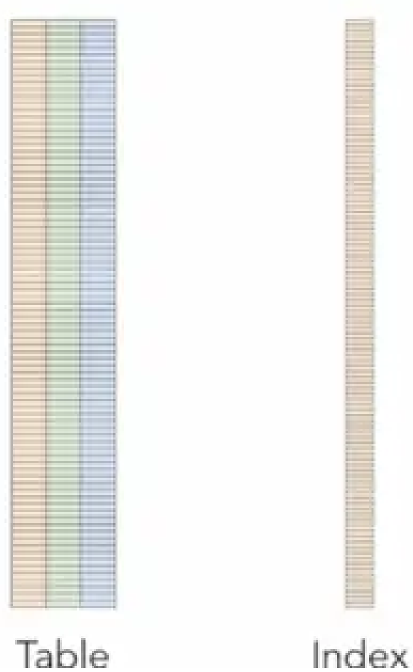
This is the basic idea behind partitioning.

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.

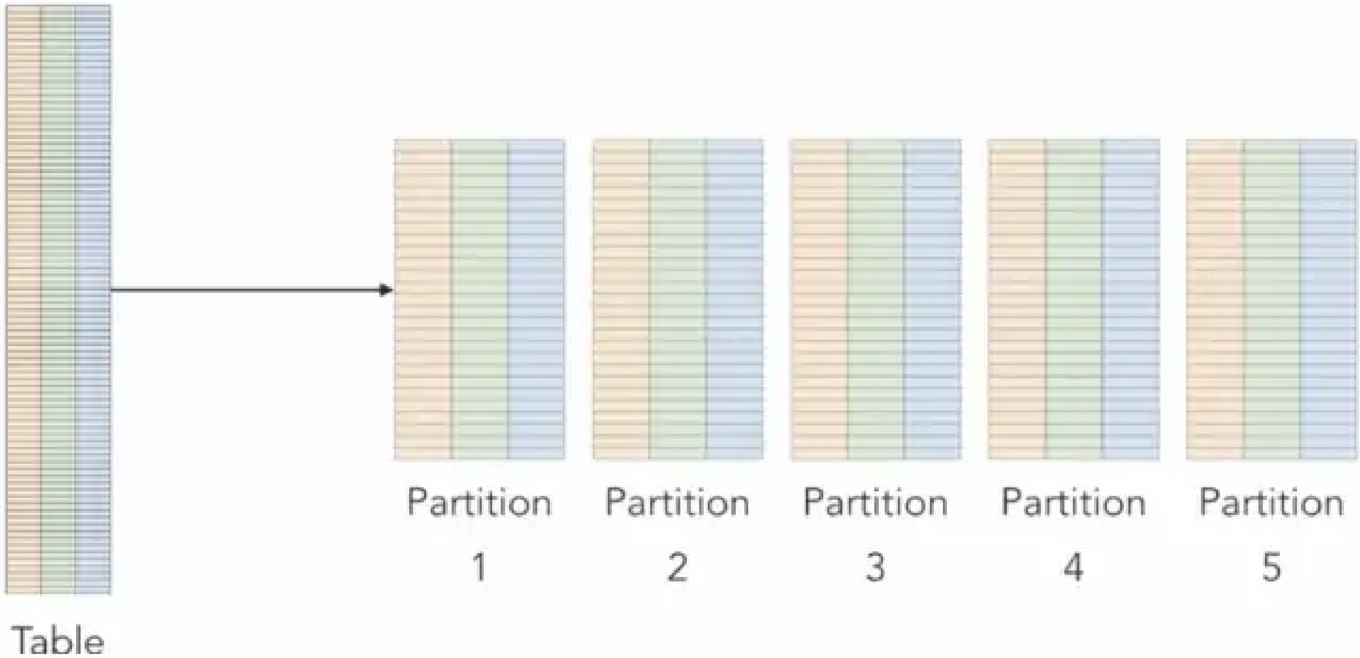
What Is Partitioning?

- 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

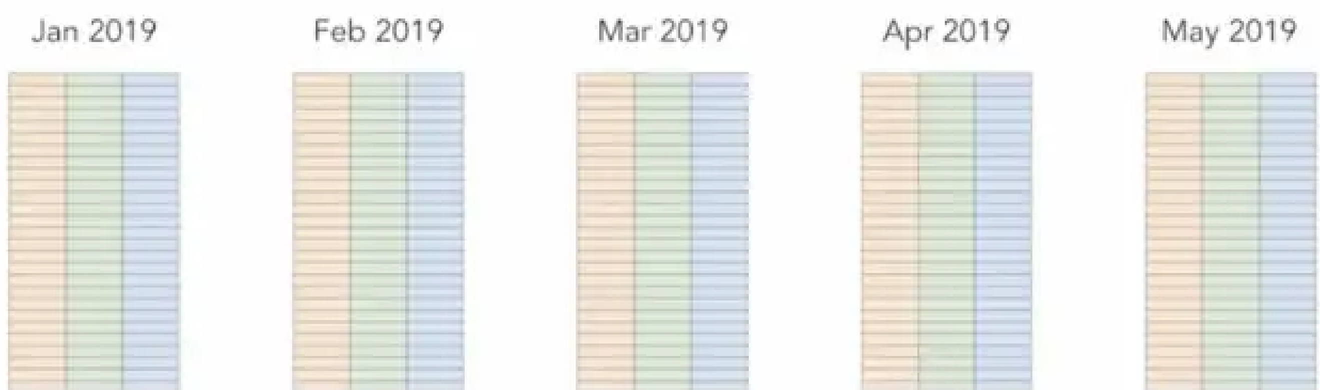
Large Tables = Large Indexes

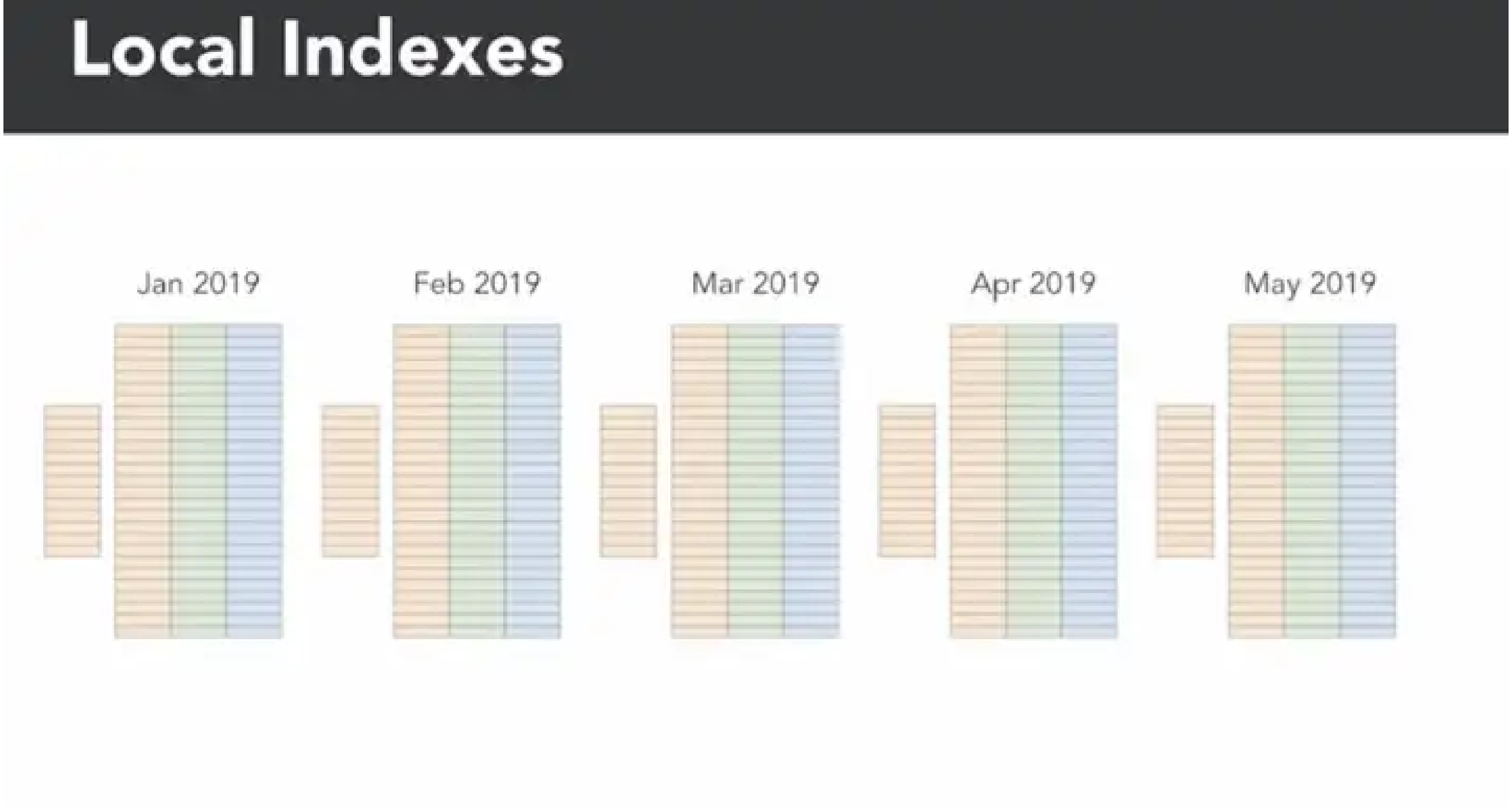


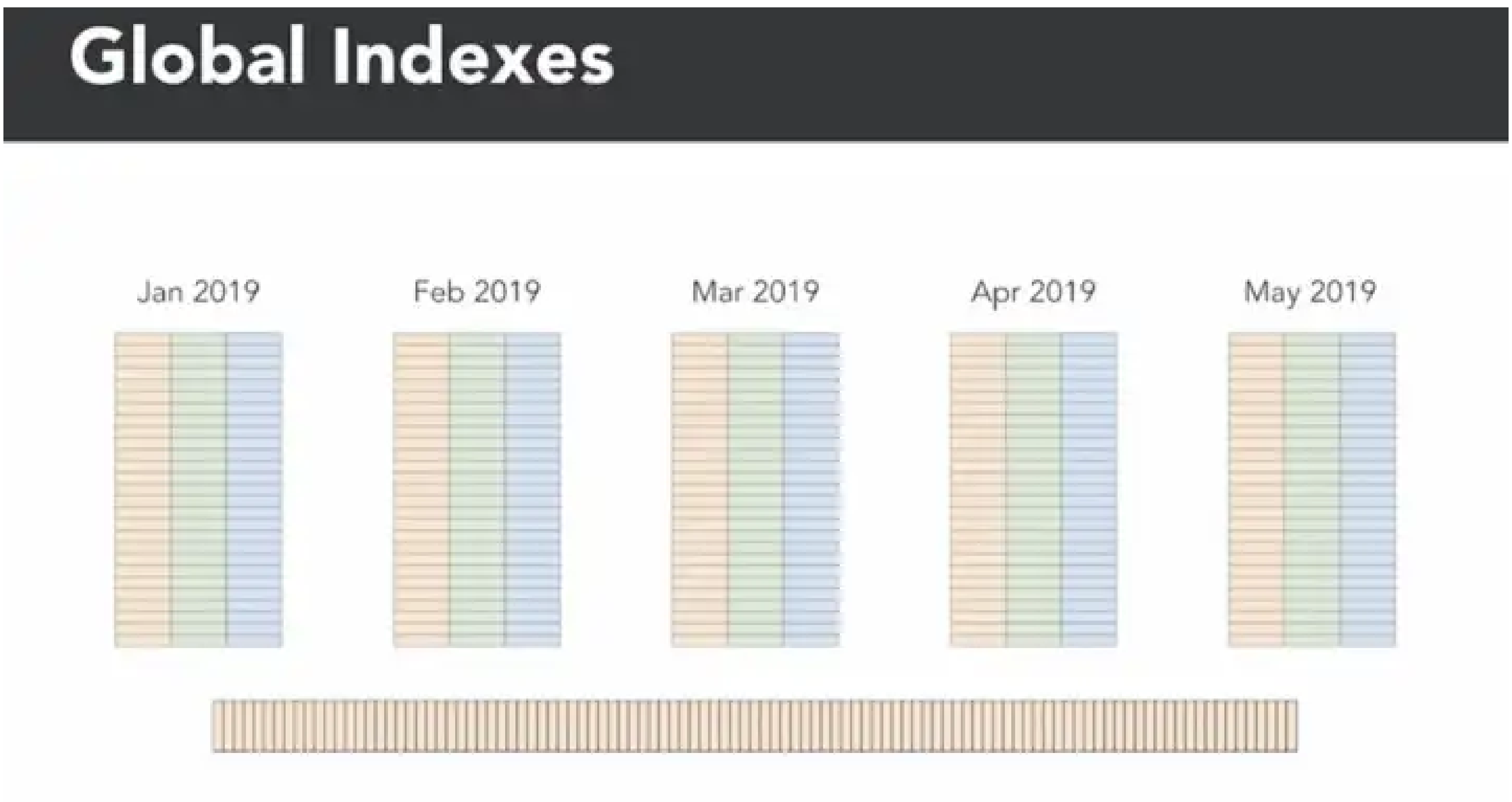
Partition



Partition Key



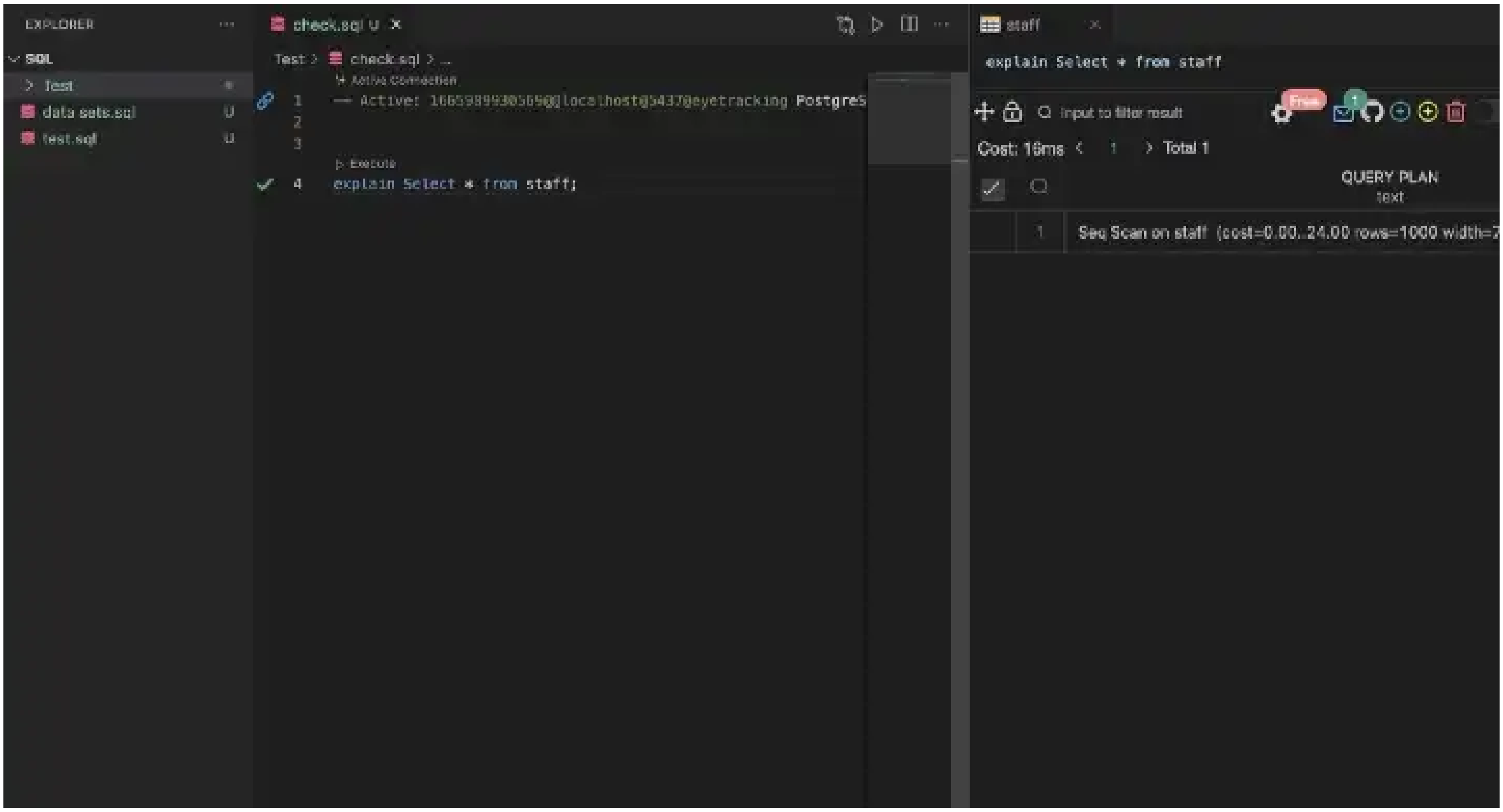




Explain and analyze

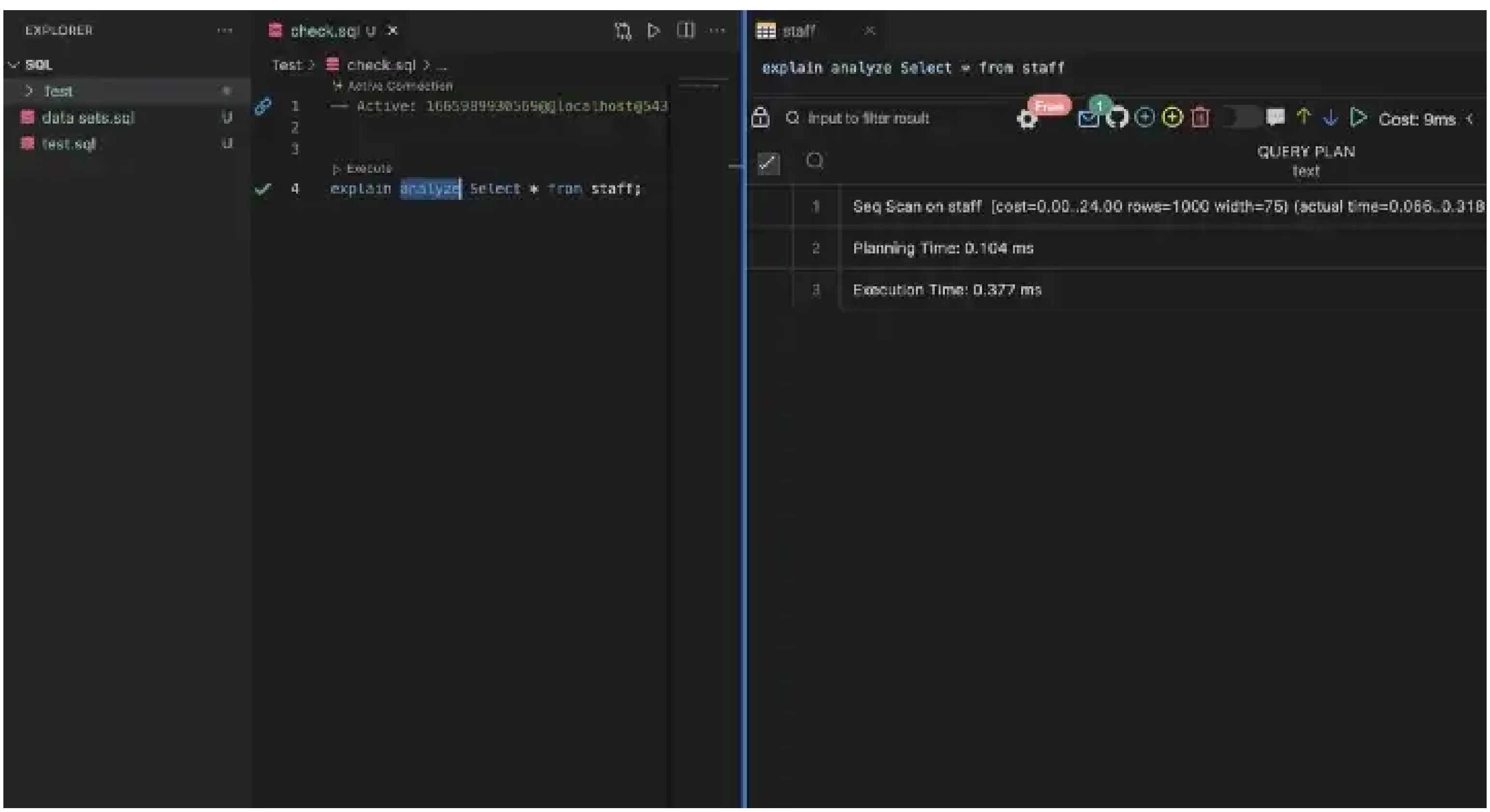
Now, one of the first steps, in optimizing queries, is understanding how they're executing. Remember, SEQUEL is declarative. So, for example, here is a simple declarative query.

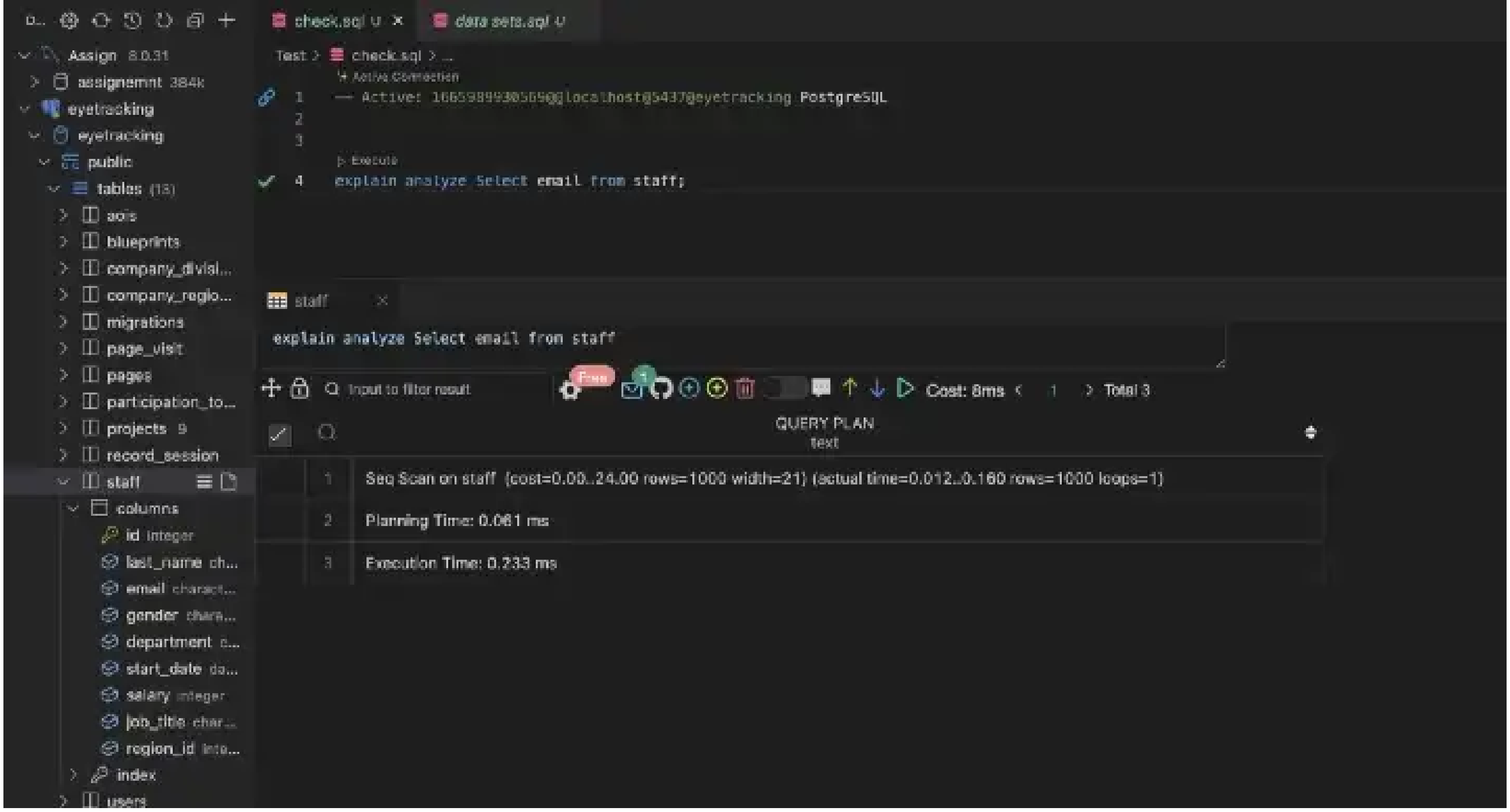
Cost is a measure of computation required to complete the step. In this example, the cost spans from 00 to 24.00.



we also see the number of rows returned, and the **width of the row**.

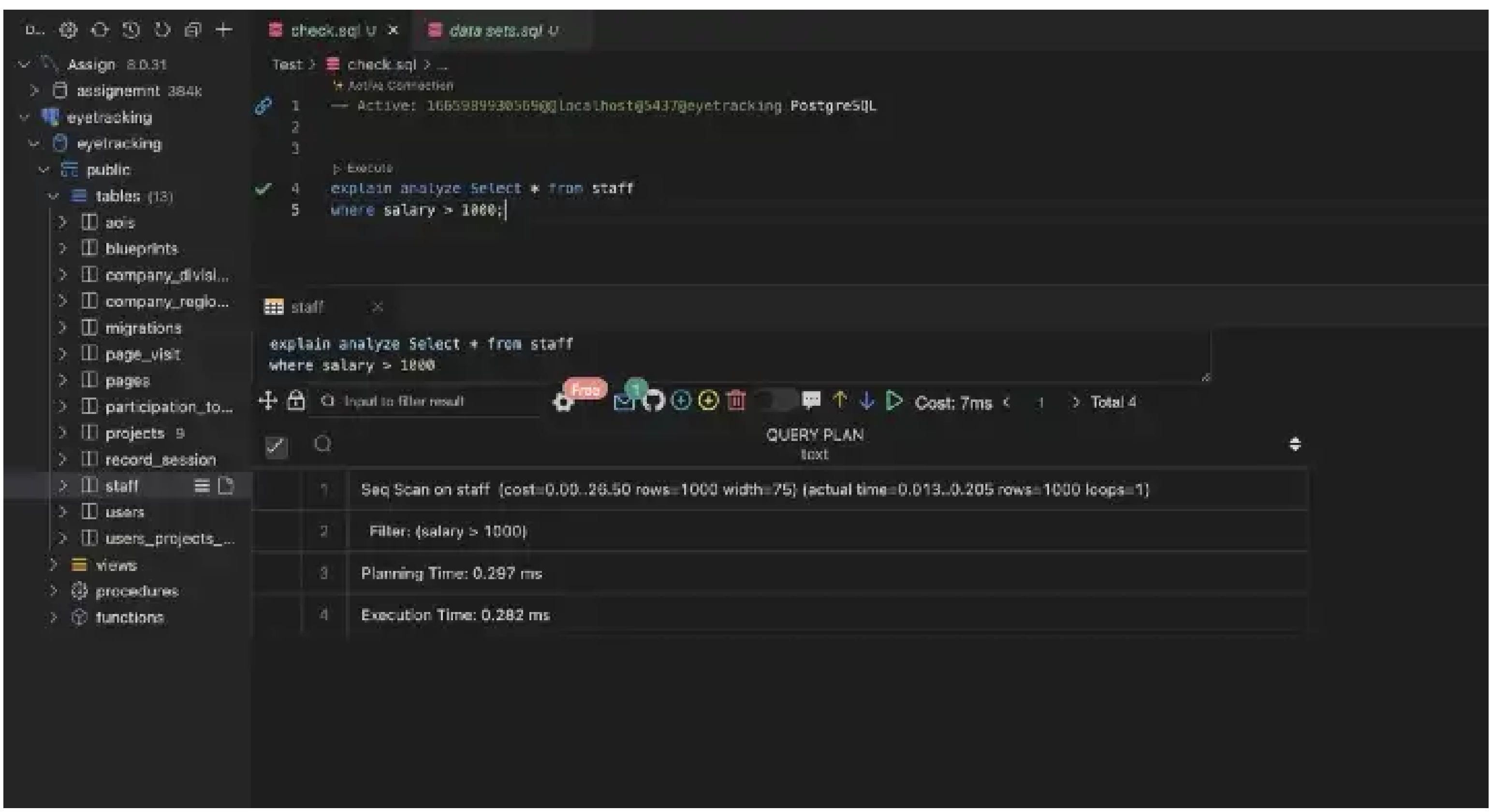
This gives us an indication of how much data is returned.



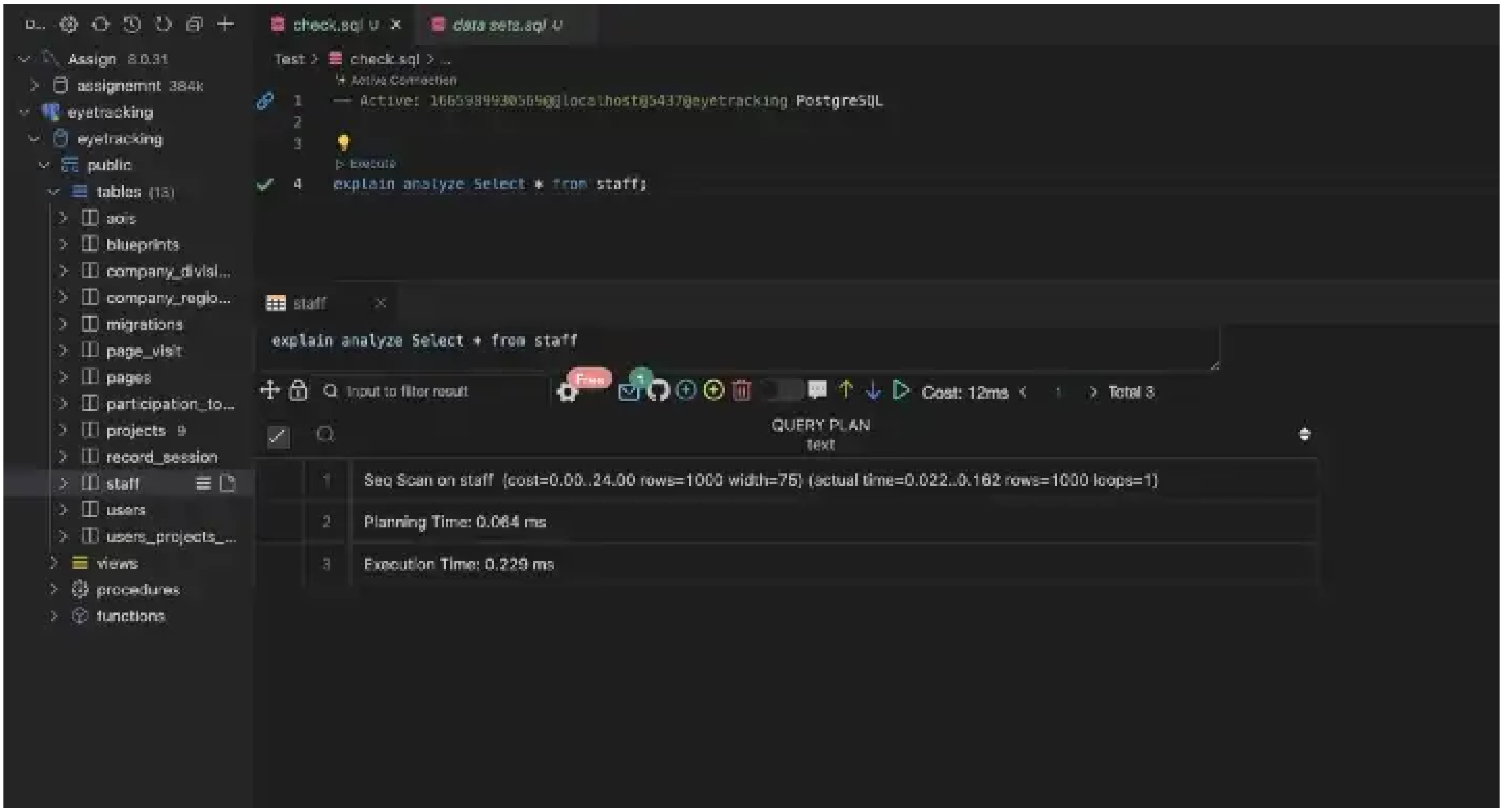


If you're working with large data sets, or are concerned about how much data is returned by a query, **you can use the row count and the width, to help guide you as you try to reduce the amount of data that's returned.**

Where clasuse (Execution Time / Computation Time)



Computation Time = 0.25.50



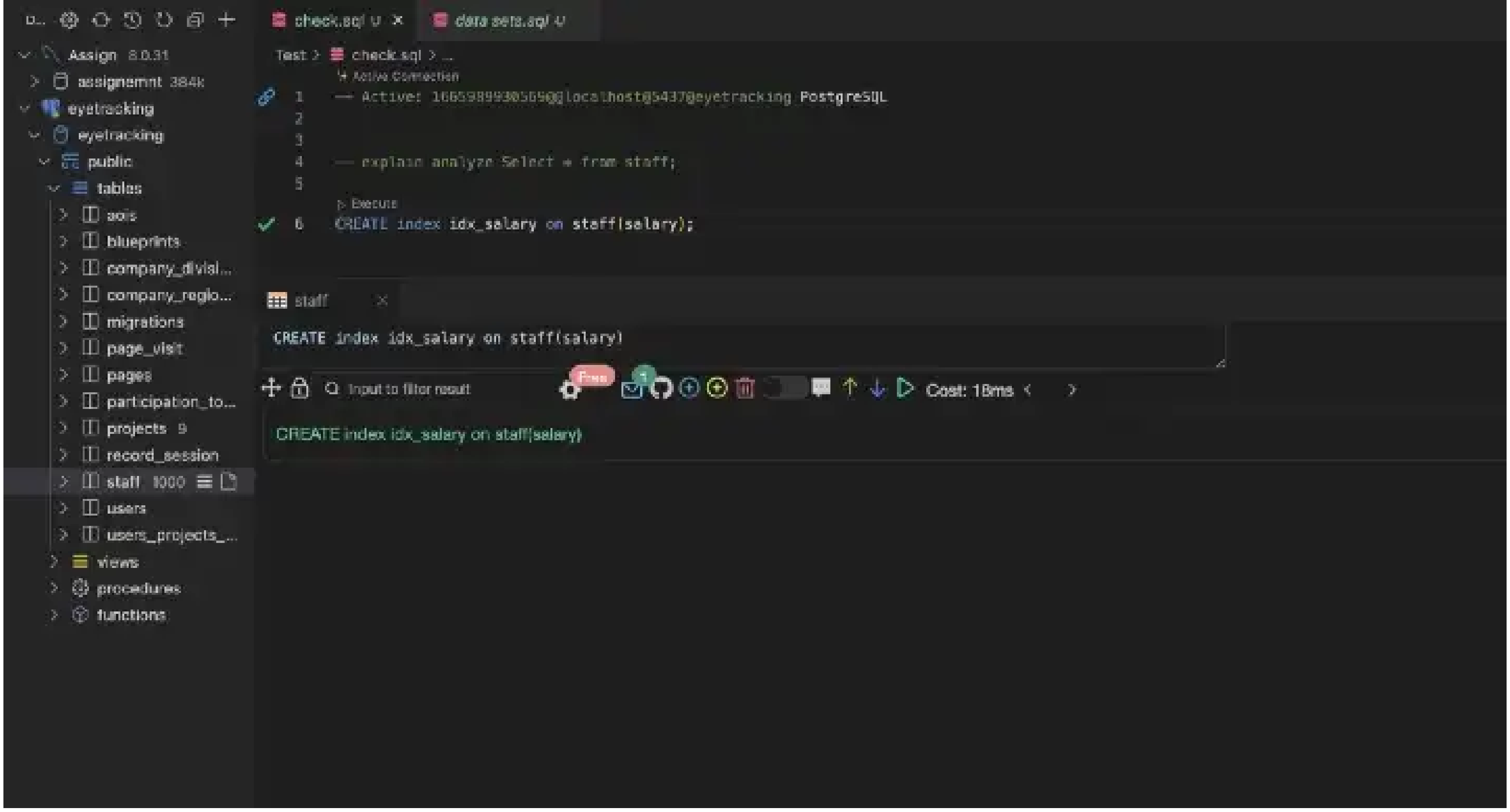
Computation Time = 0.24.00

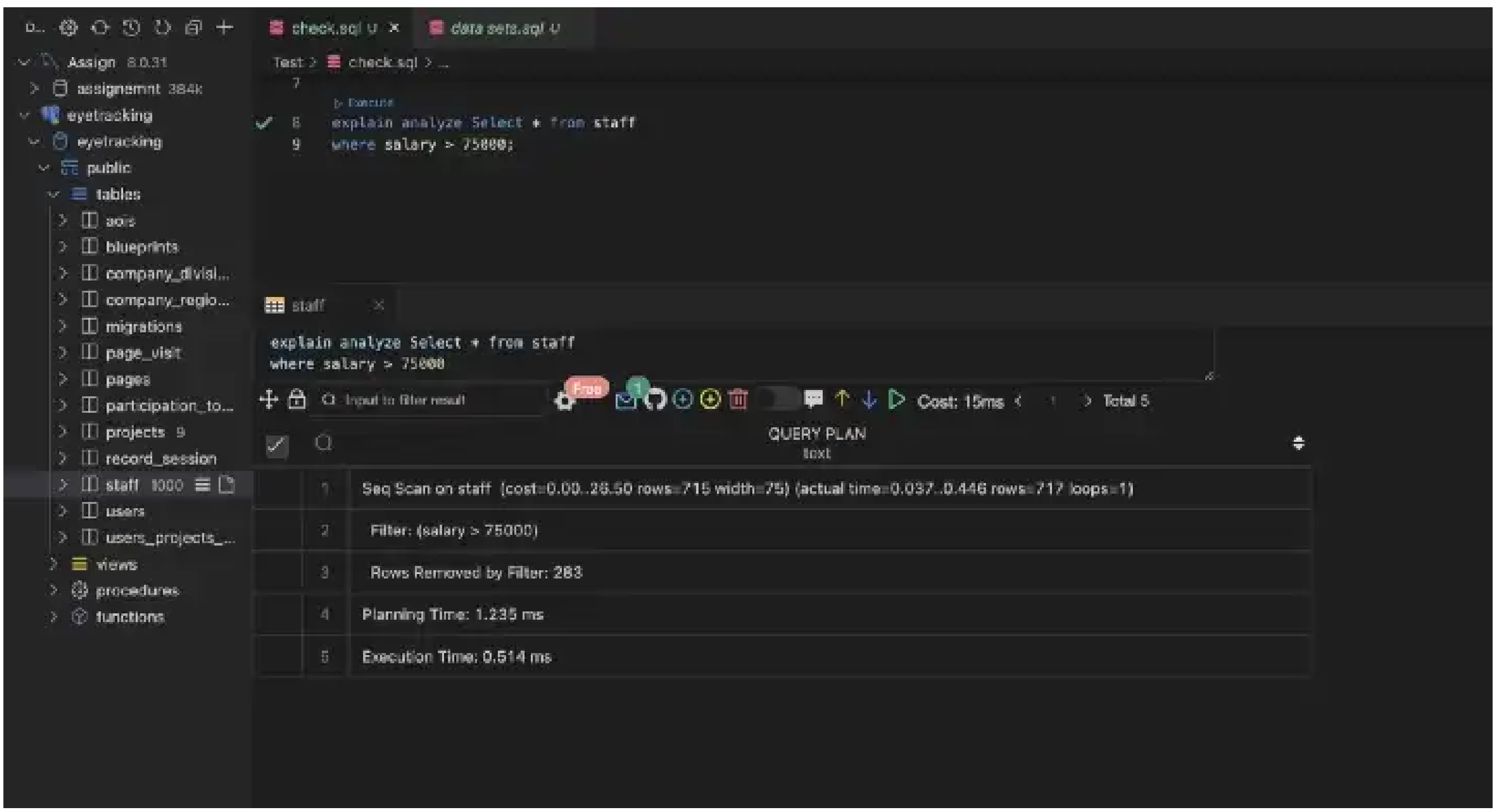
Conslusion : So the total estimated time can actually be less when returning more data if there are fewer steps in the execution plan.

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.

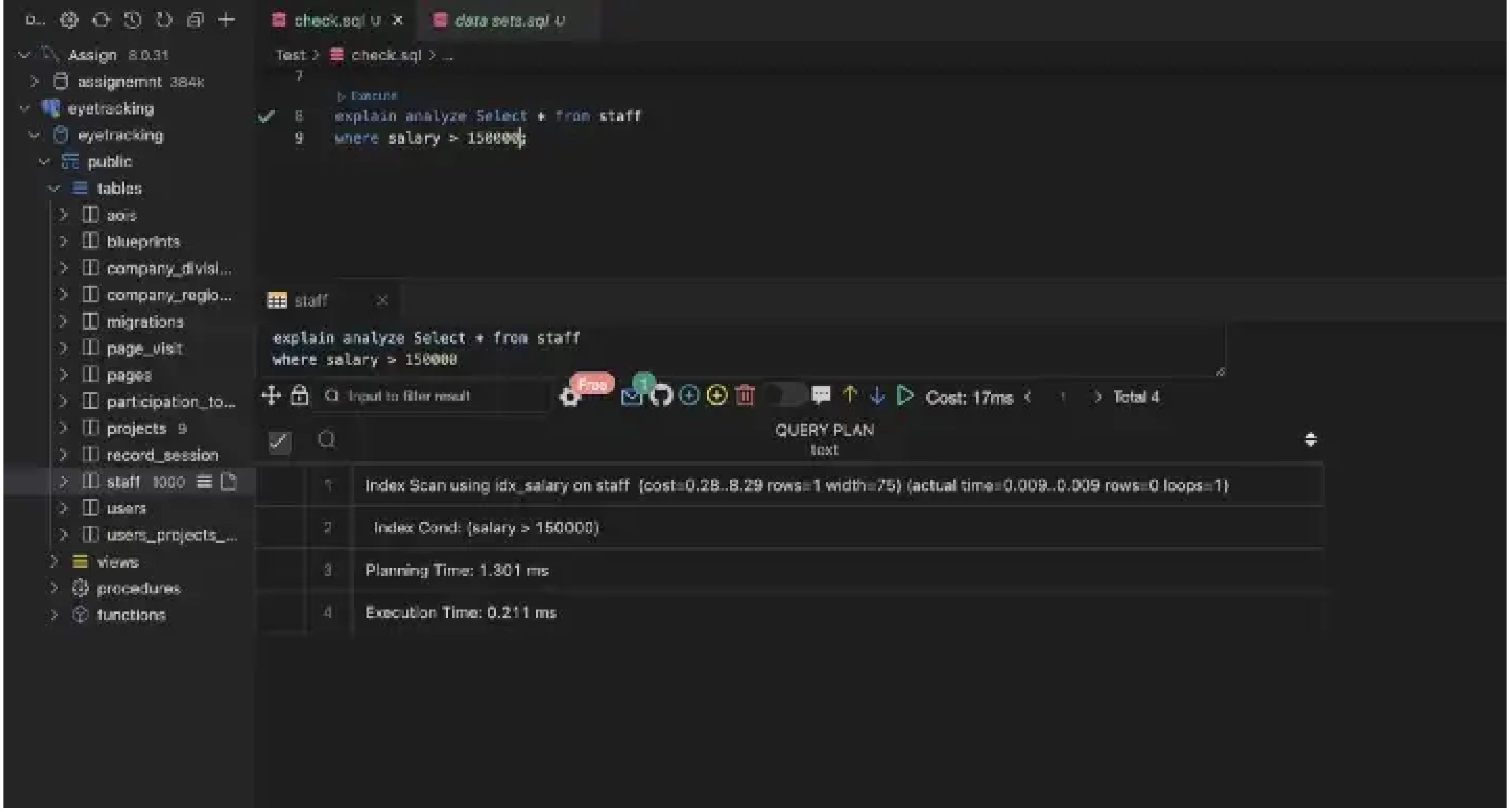
One of the nice things about indexes is that you can include multiple columns, But for now we are doing Indexing on salary.





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.

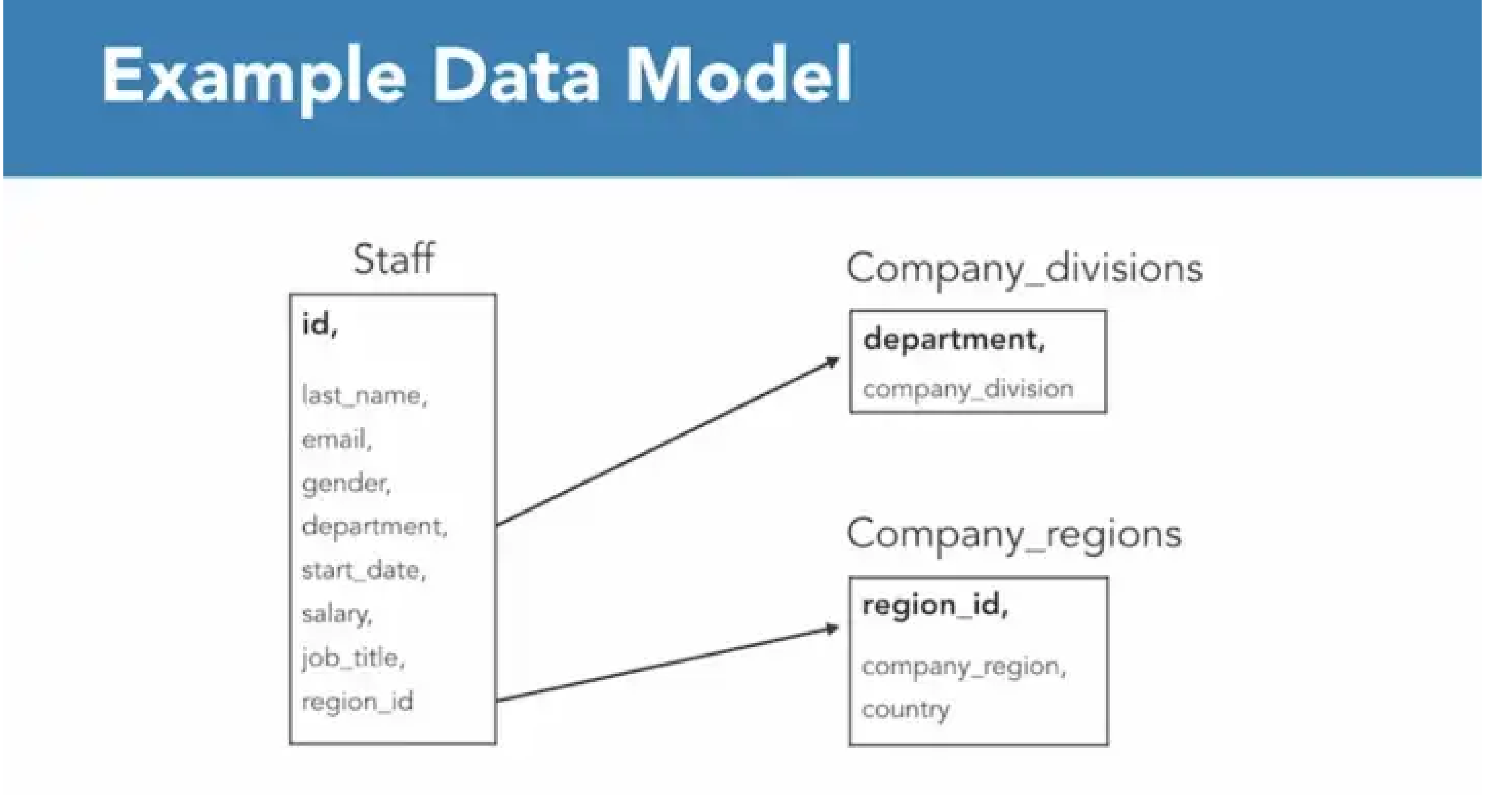


Type of Index

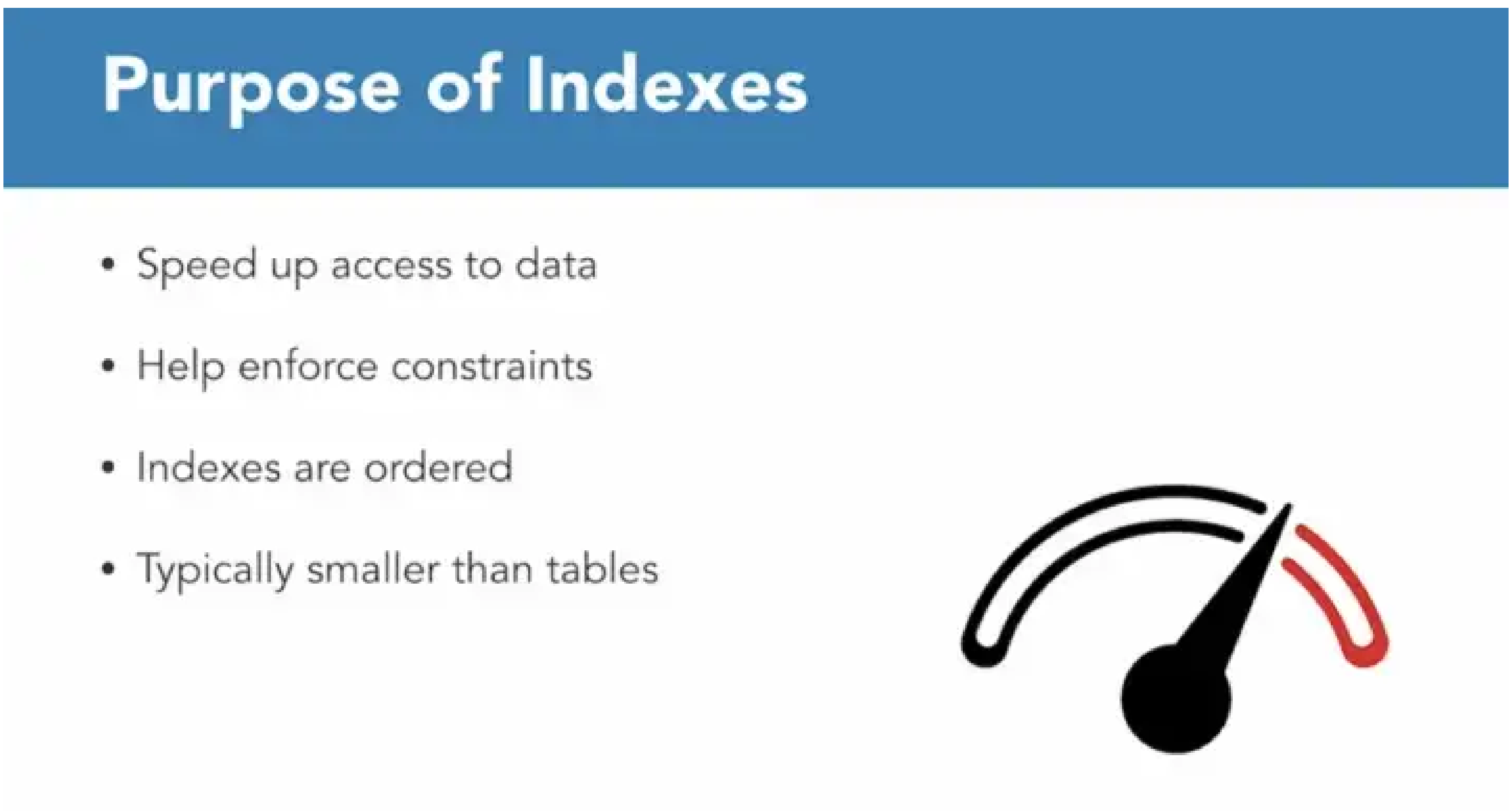
We'll start our learning of indexes by looking at a simple data model.

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.



Now what is the purpose of indexes



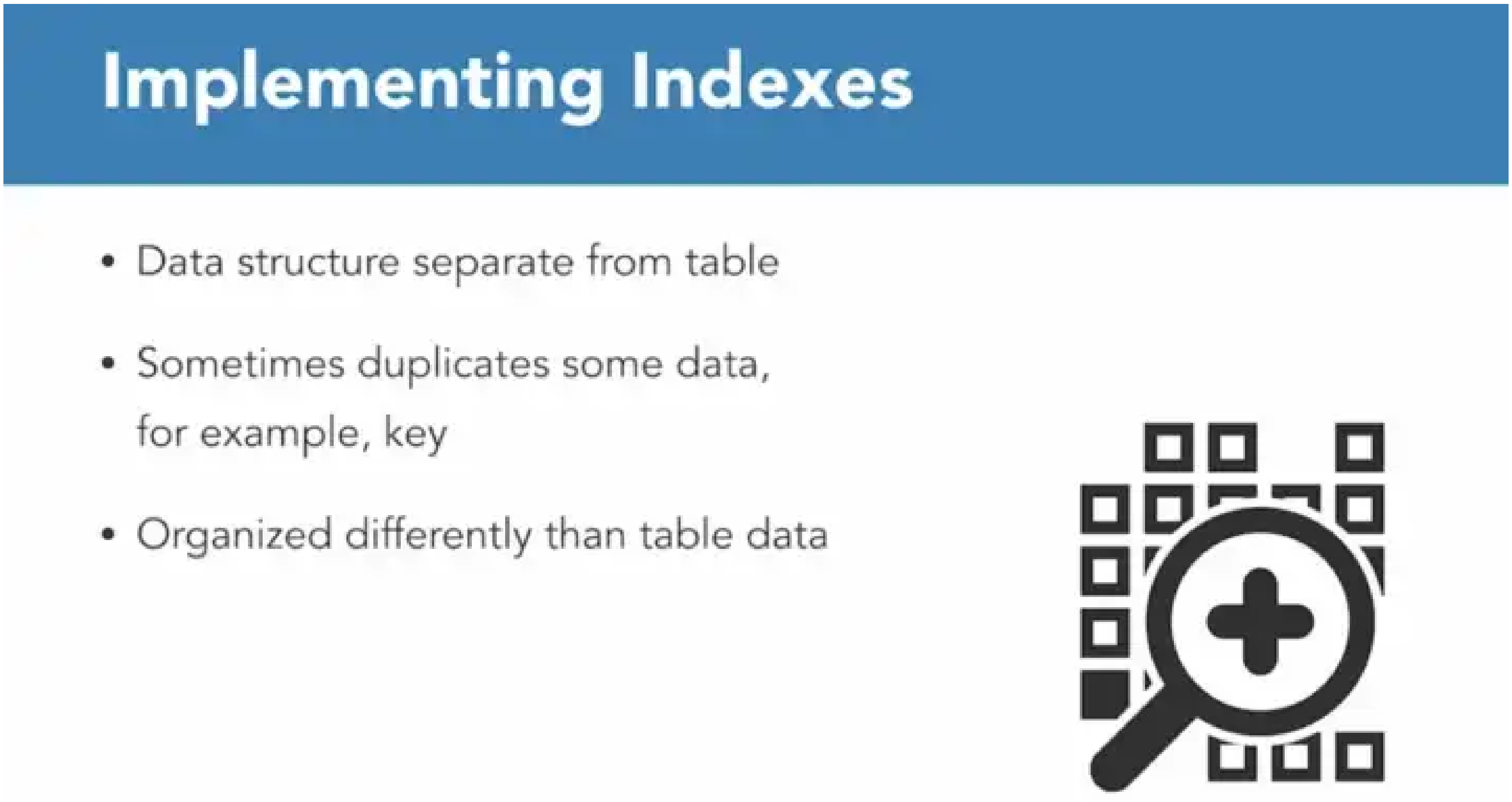
So, Inshort Index reduce scanning :)

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.

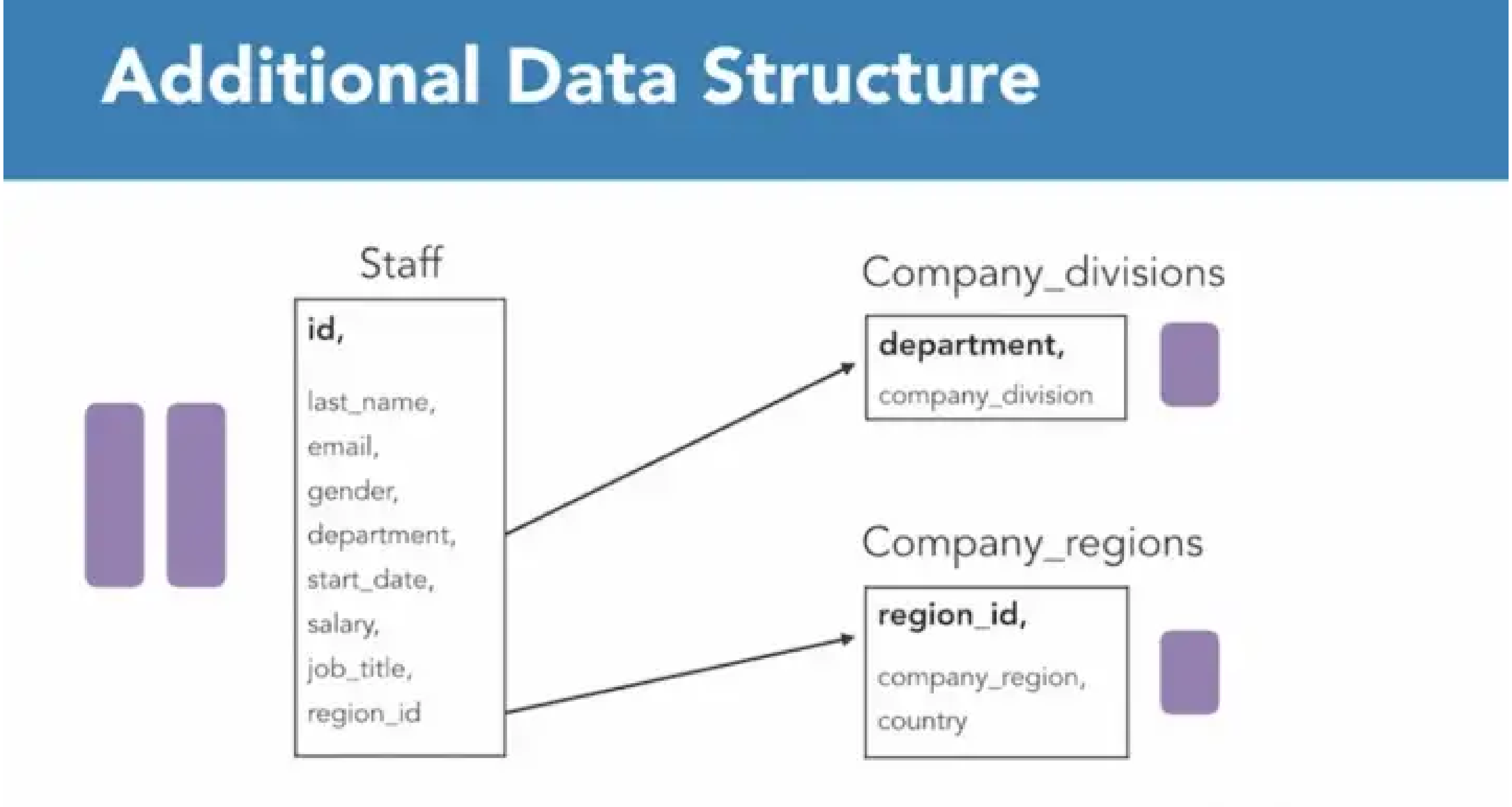
That's great news for querying, because reading data from memory is much faster than reading data from a hard disk, or even from solid state drives, or SSDs.

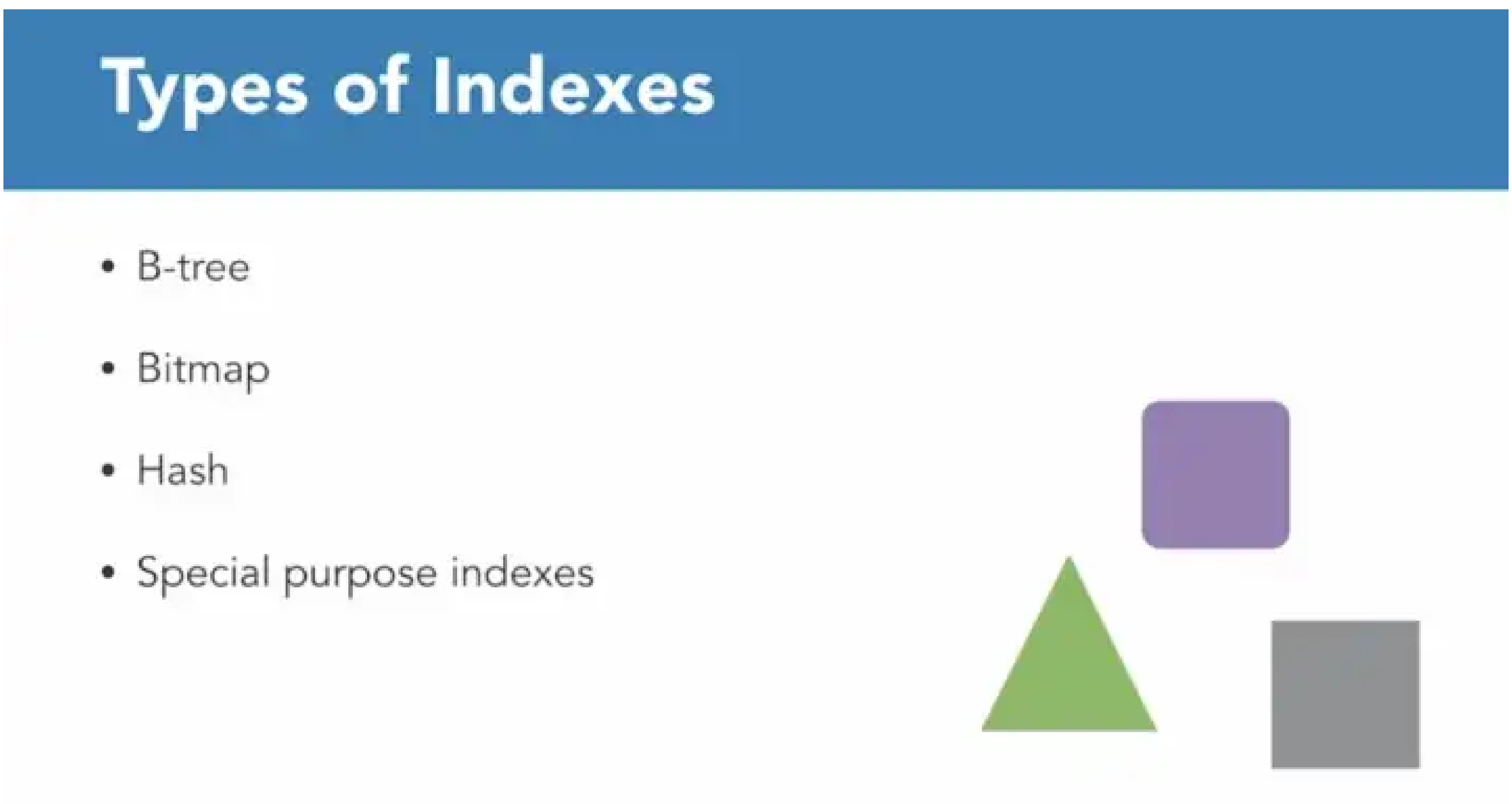


Indexes use storage space in addition to what is in the table. Now usually, we try not to duplicate data. But indexes are a special case.

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.



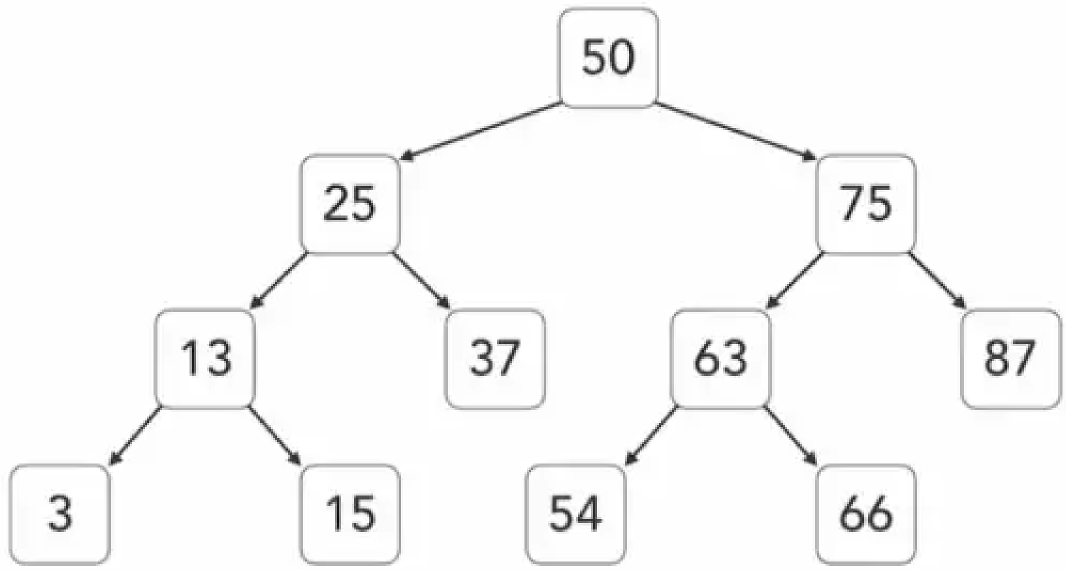


B-Tree Indexes

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.

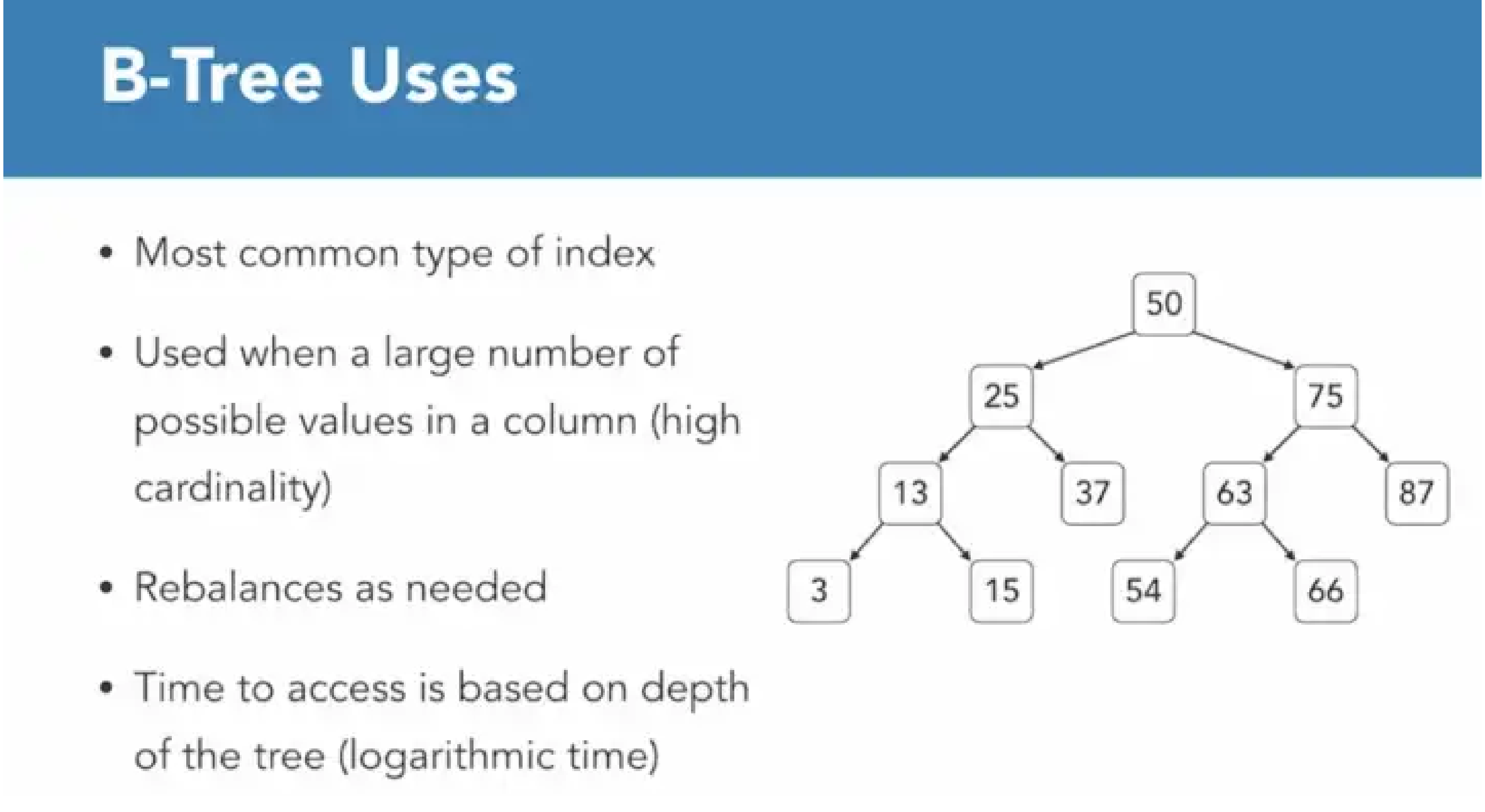
It's used when there are a large number of distinct values in a column.

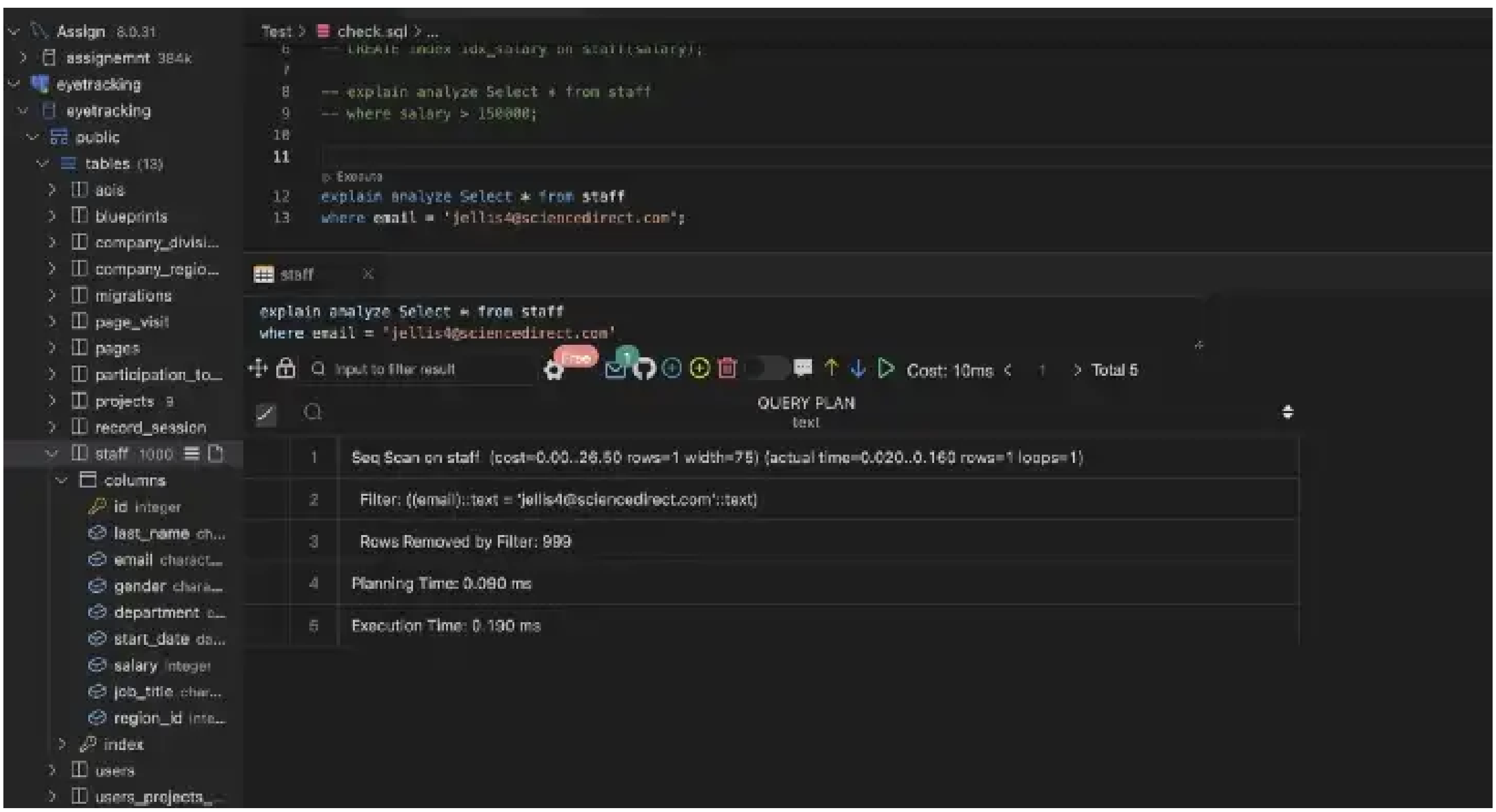
This is called high cardinality.

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

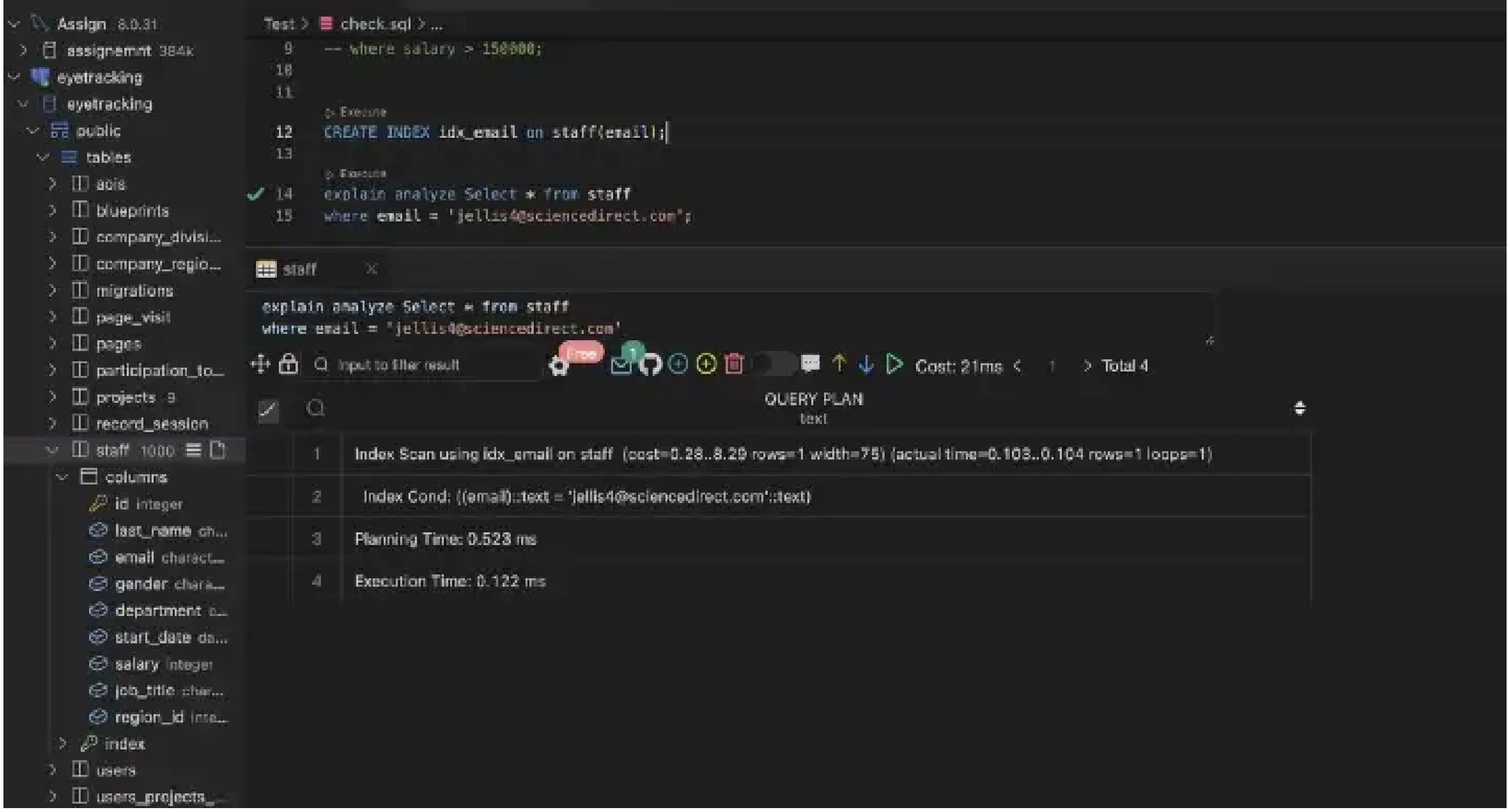
This prevents a lopsided tree that would be fast to search on one side and slower on the other.

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.



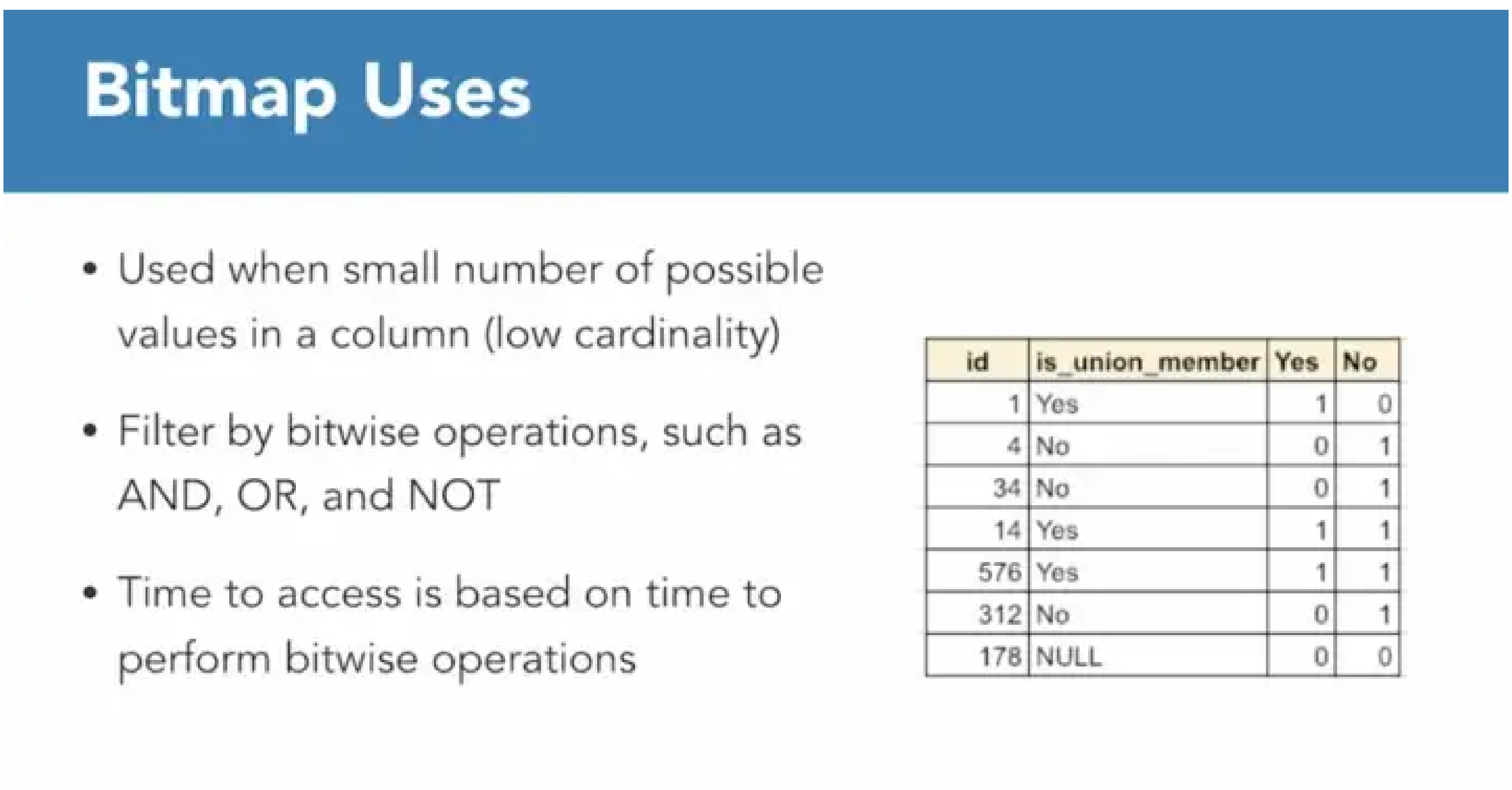


Now B tree is a default type of index, so we can use a basic create index command without specifying the B tree specifically.



BitMap Index

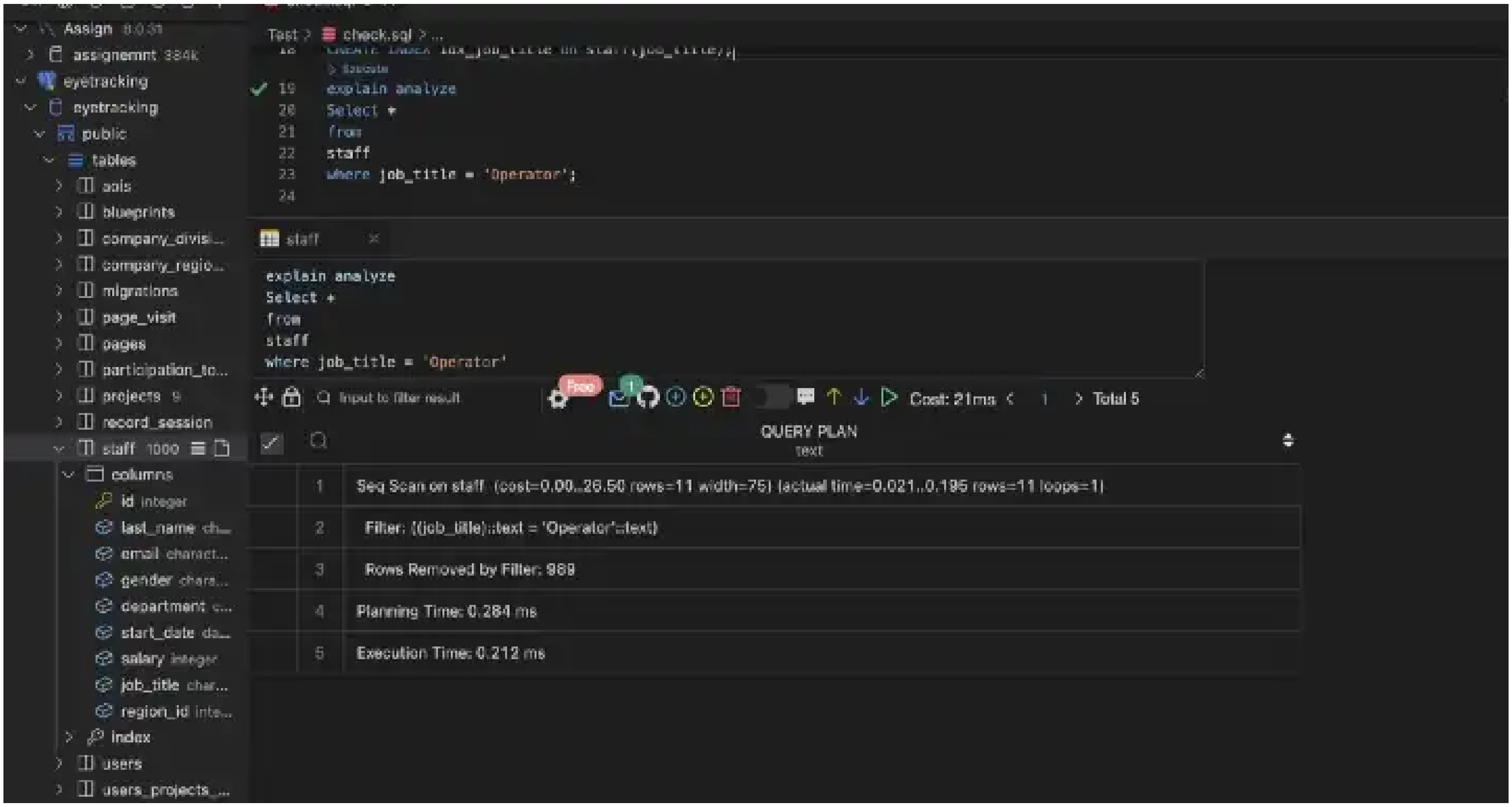
Bitmap Indexes store a series of bits for indexed values. The number of bits used is the same as the number of distinct values in a column.



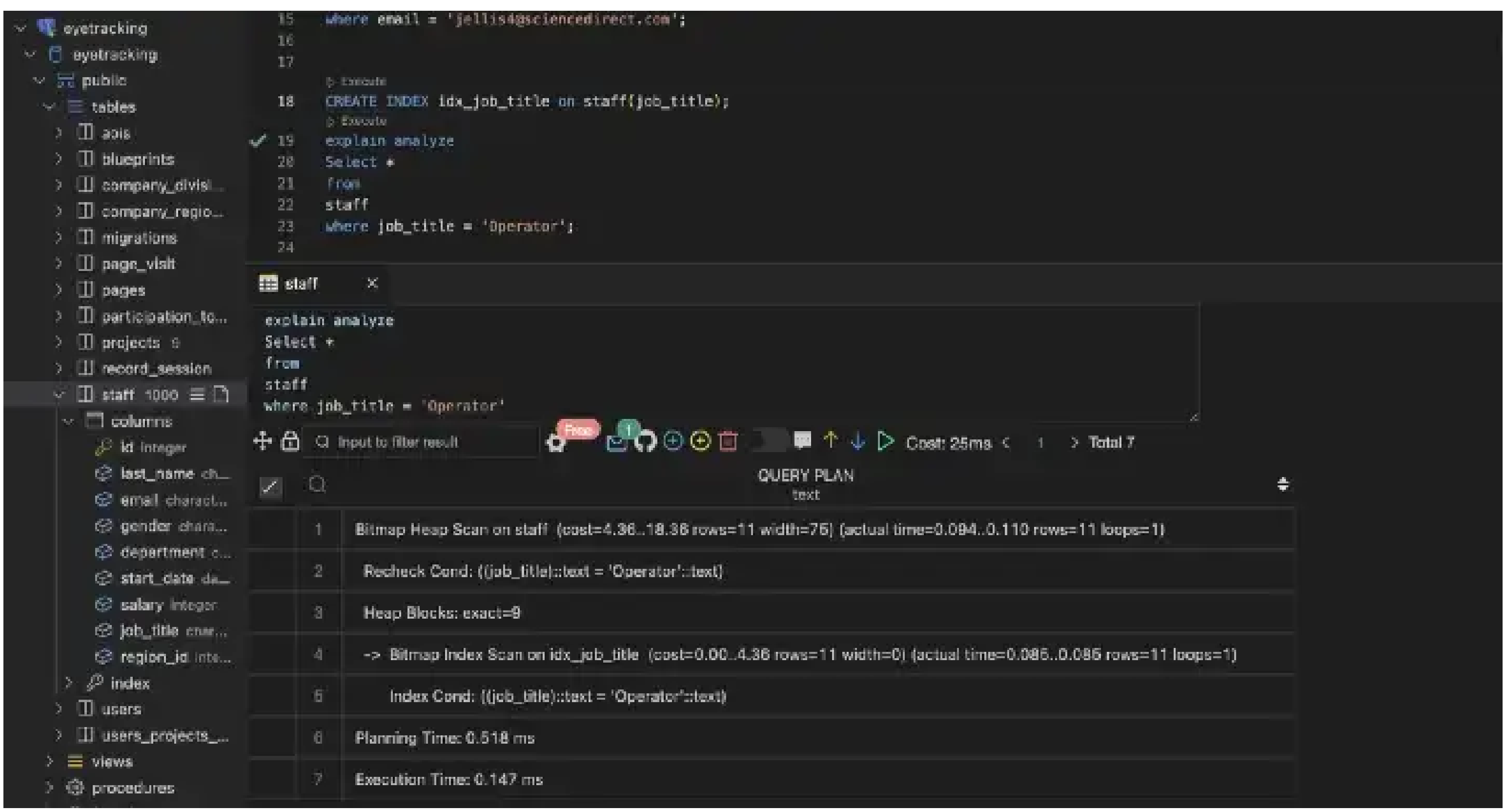
While Bitmap operations are fast, updating Bitmap Indexes can be more time consuming than other indexes.

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



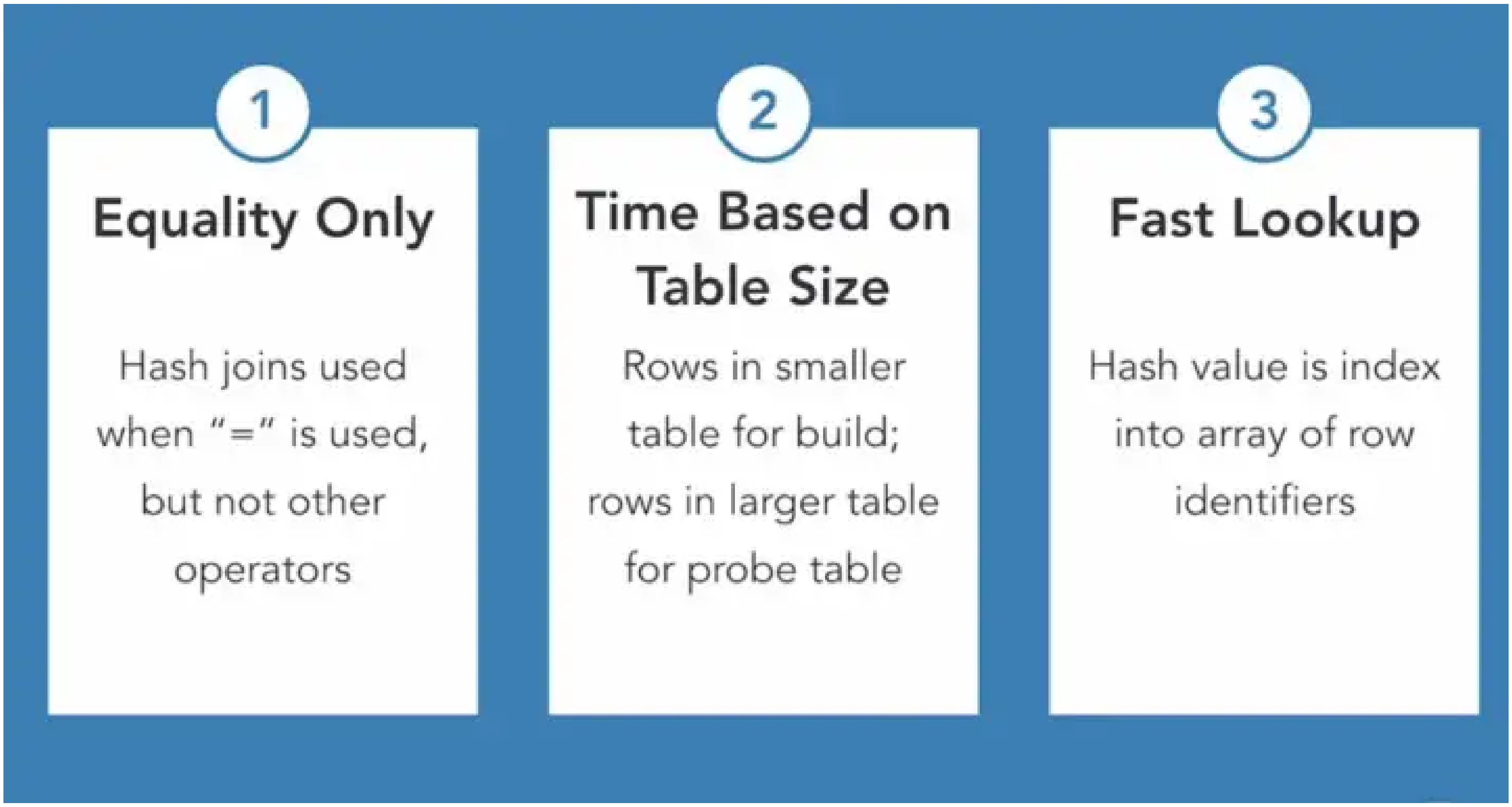
After



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

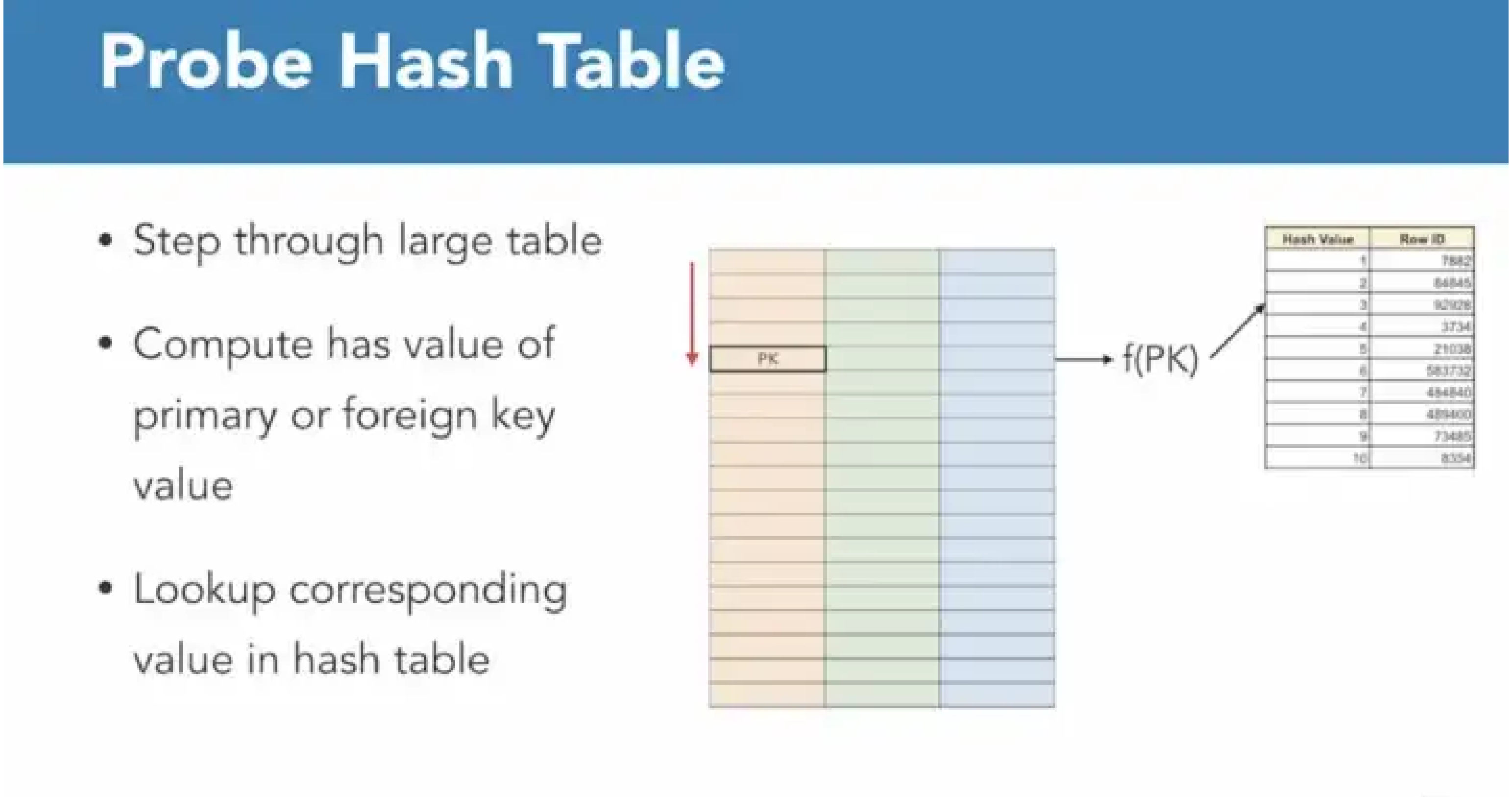


Hash joins use a function to map data into a value that can act as an index into an array.

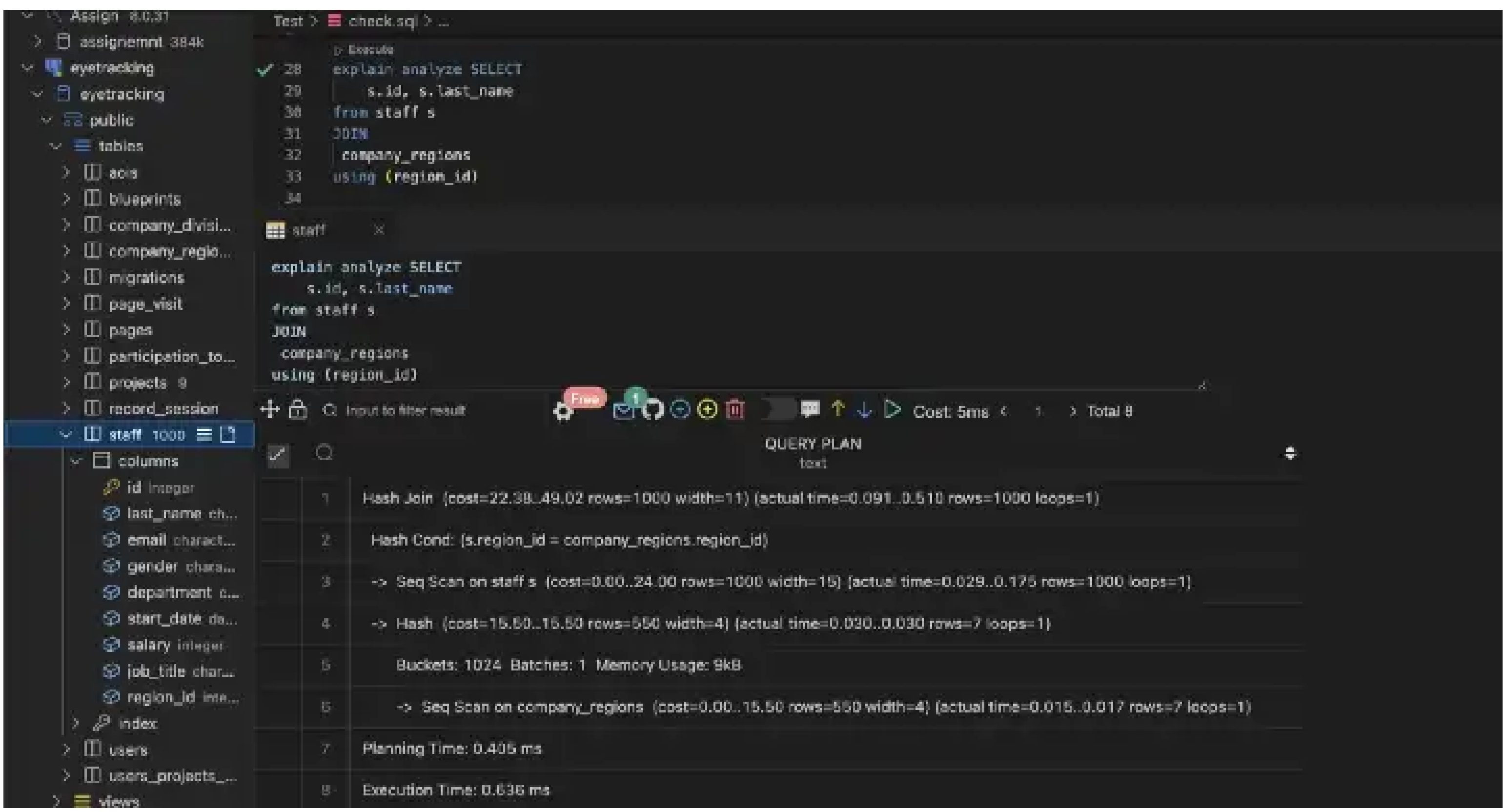
Hash functions are designed so that the output value is virtually unique across all of the values in the domain, where that is the set of possible input values.

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.



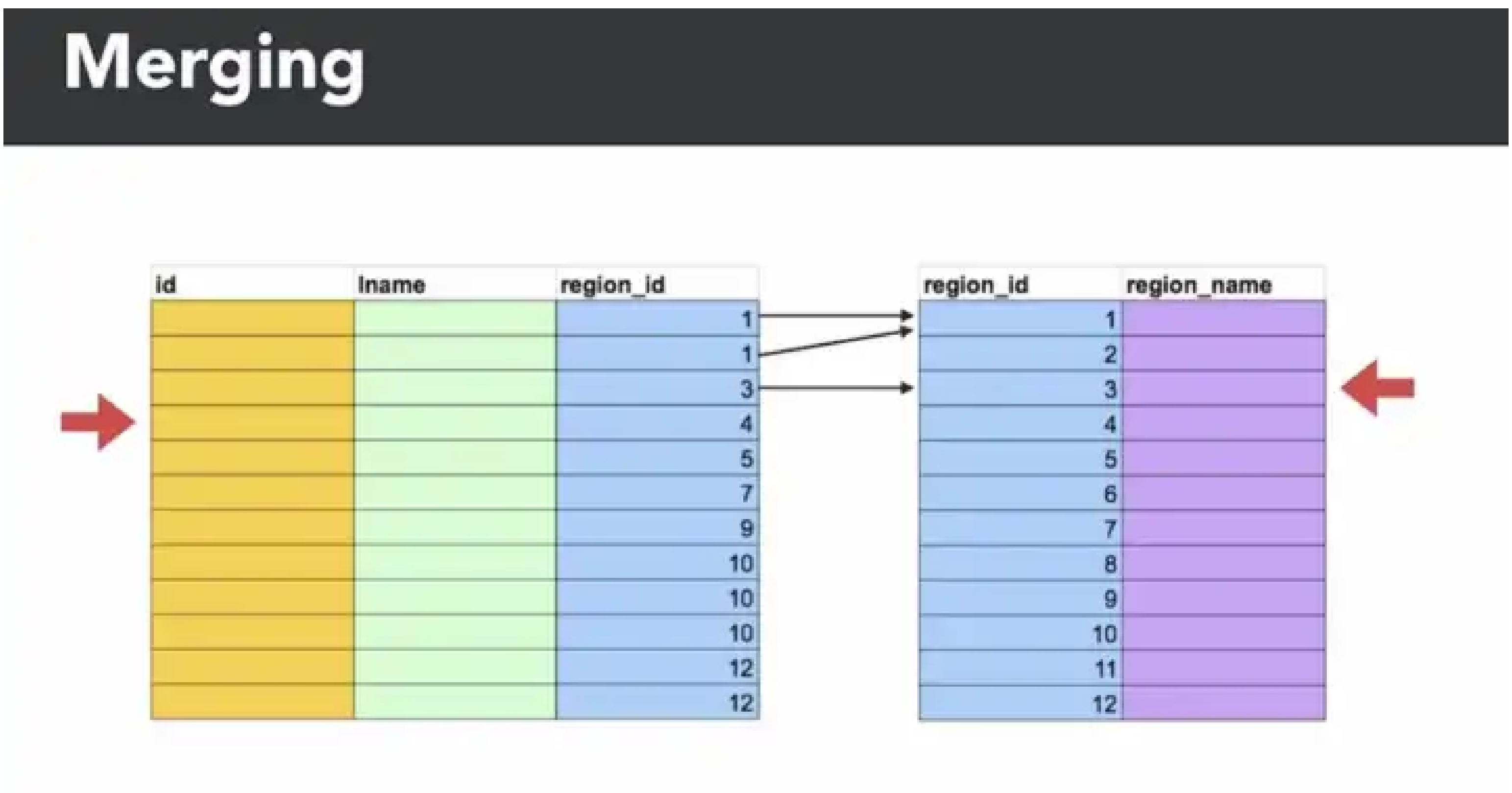
The smaller of the two tables is used. For each key value in the smaller table, we compute a hash value and store that in the table.

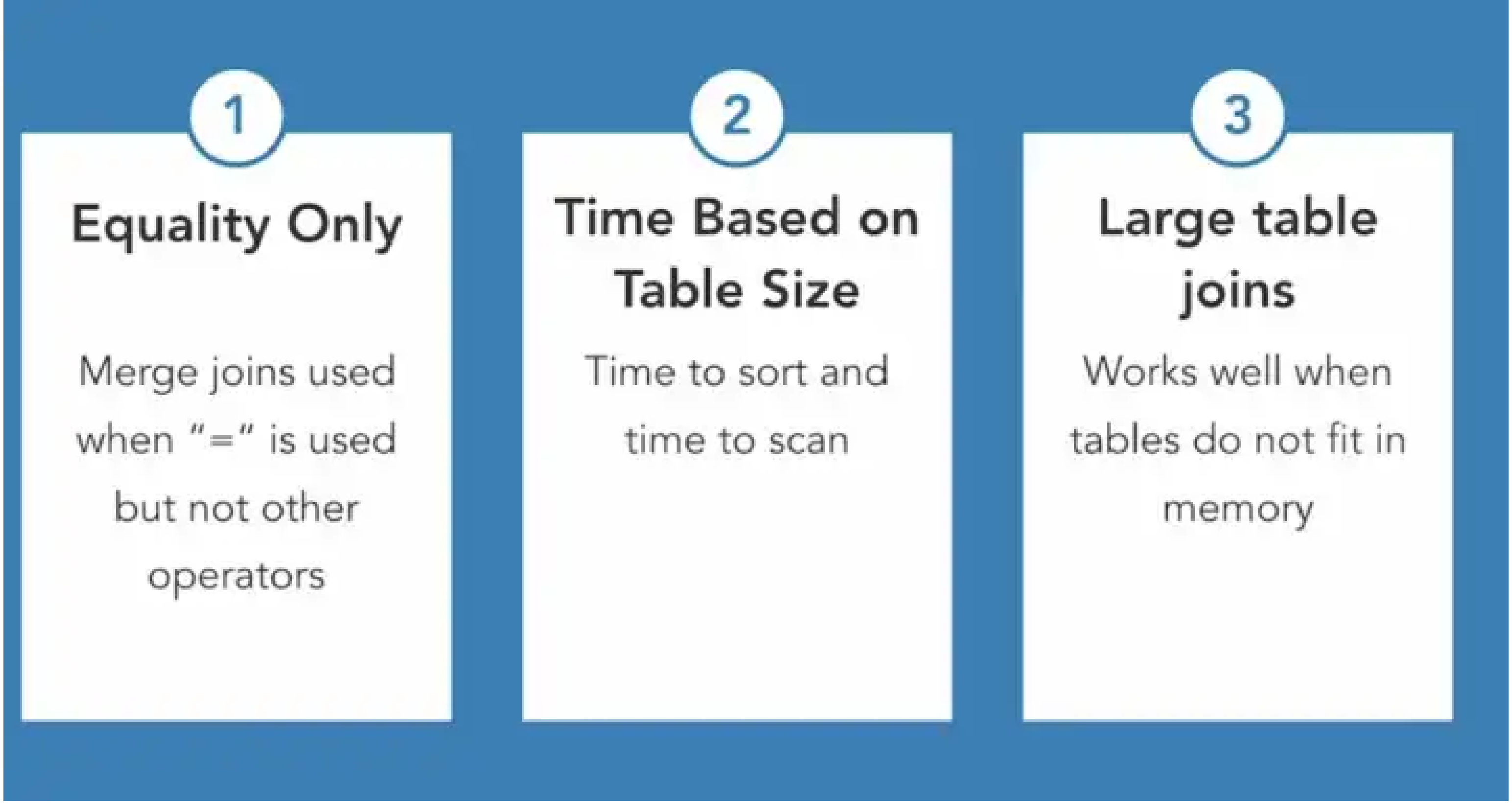


Merge Join

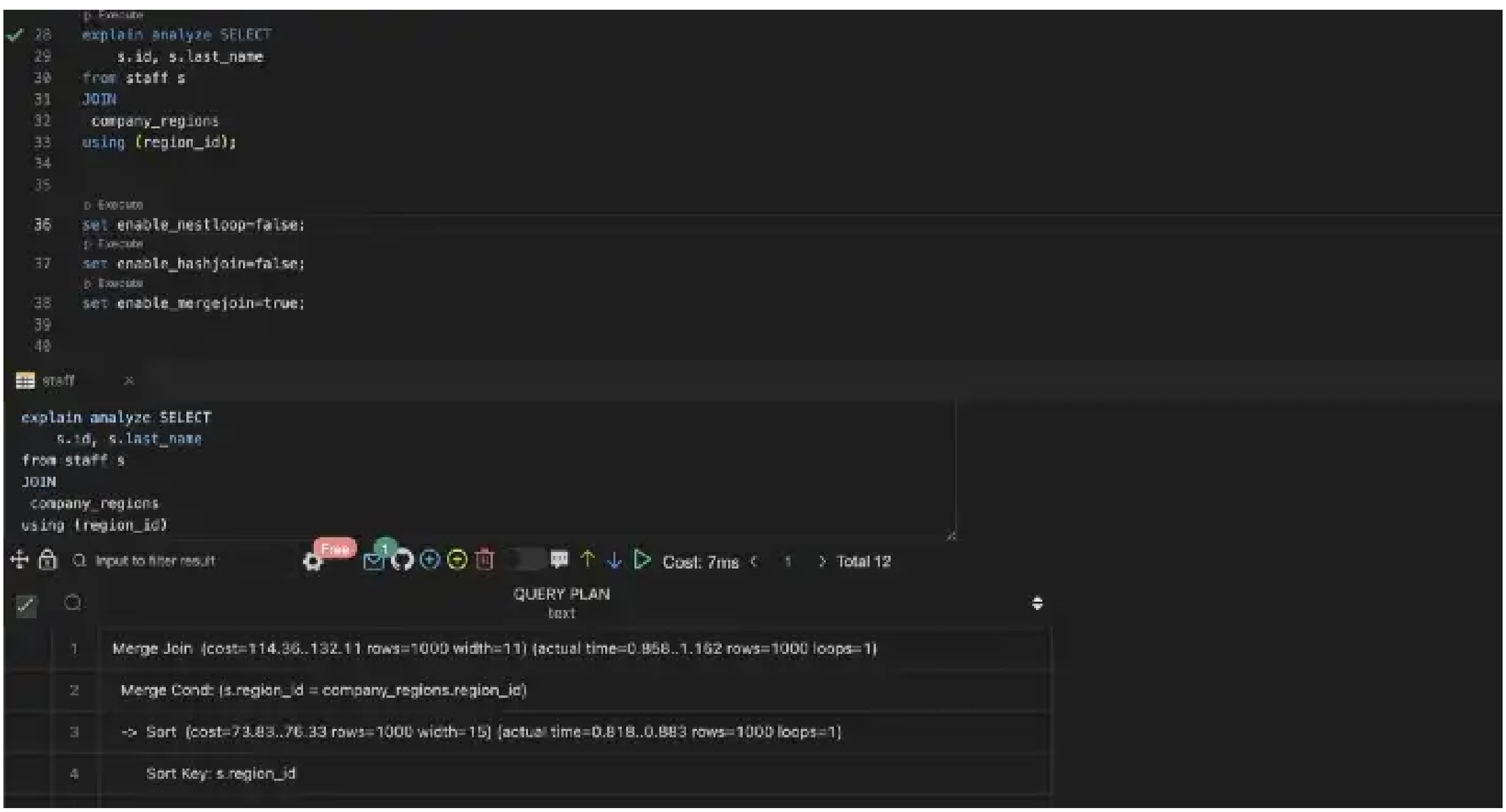
Merge joins are especially useful when tables don't fit into memory, because we can do fewer reads from disks than we would likely have to do if we used a nested loop join.

Merge joins are used only with equality join operations





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



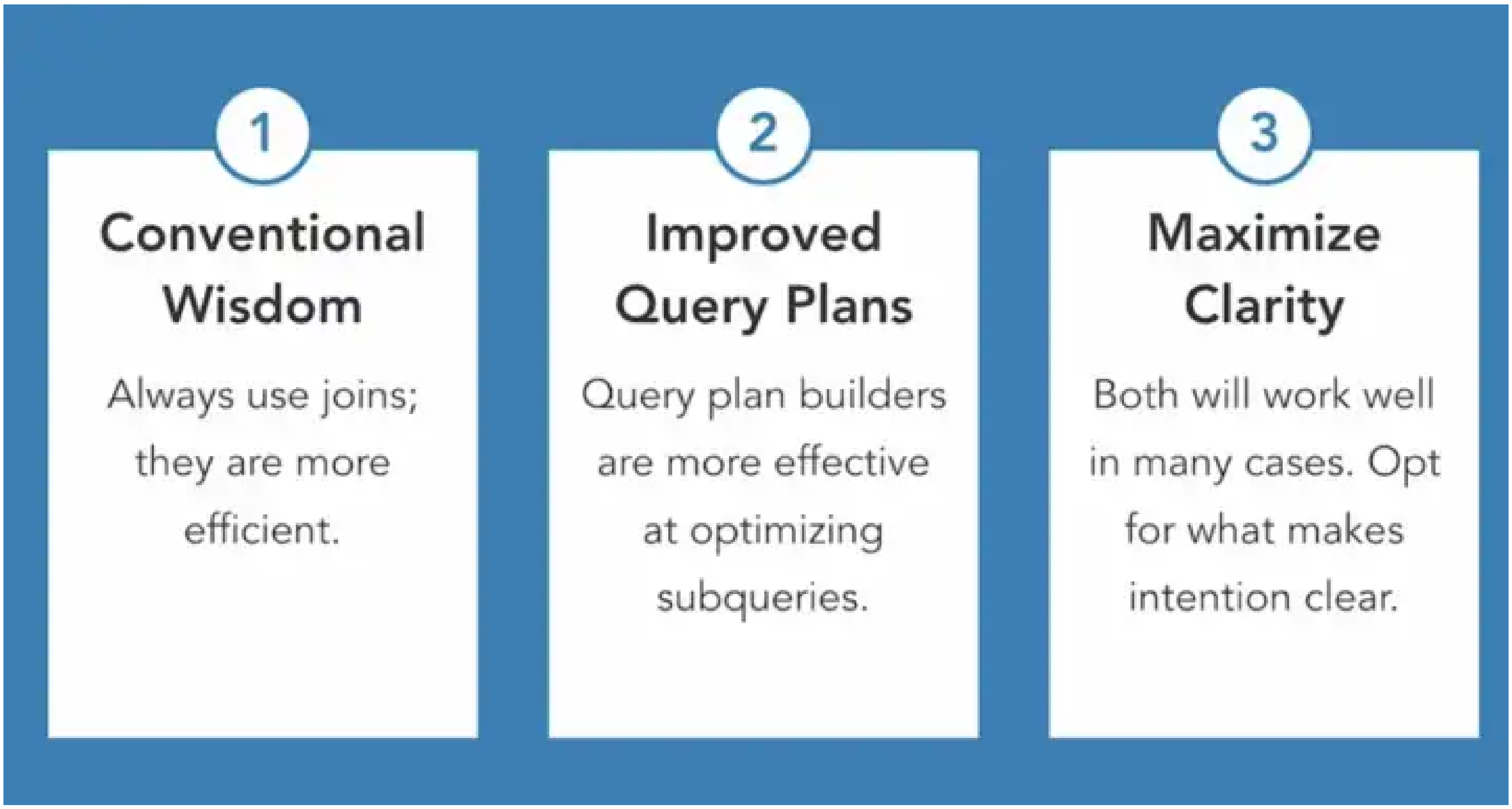
SubQueries Vs Join

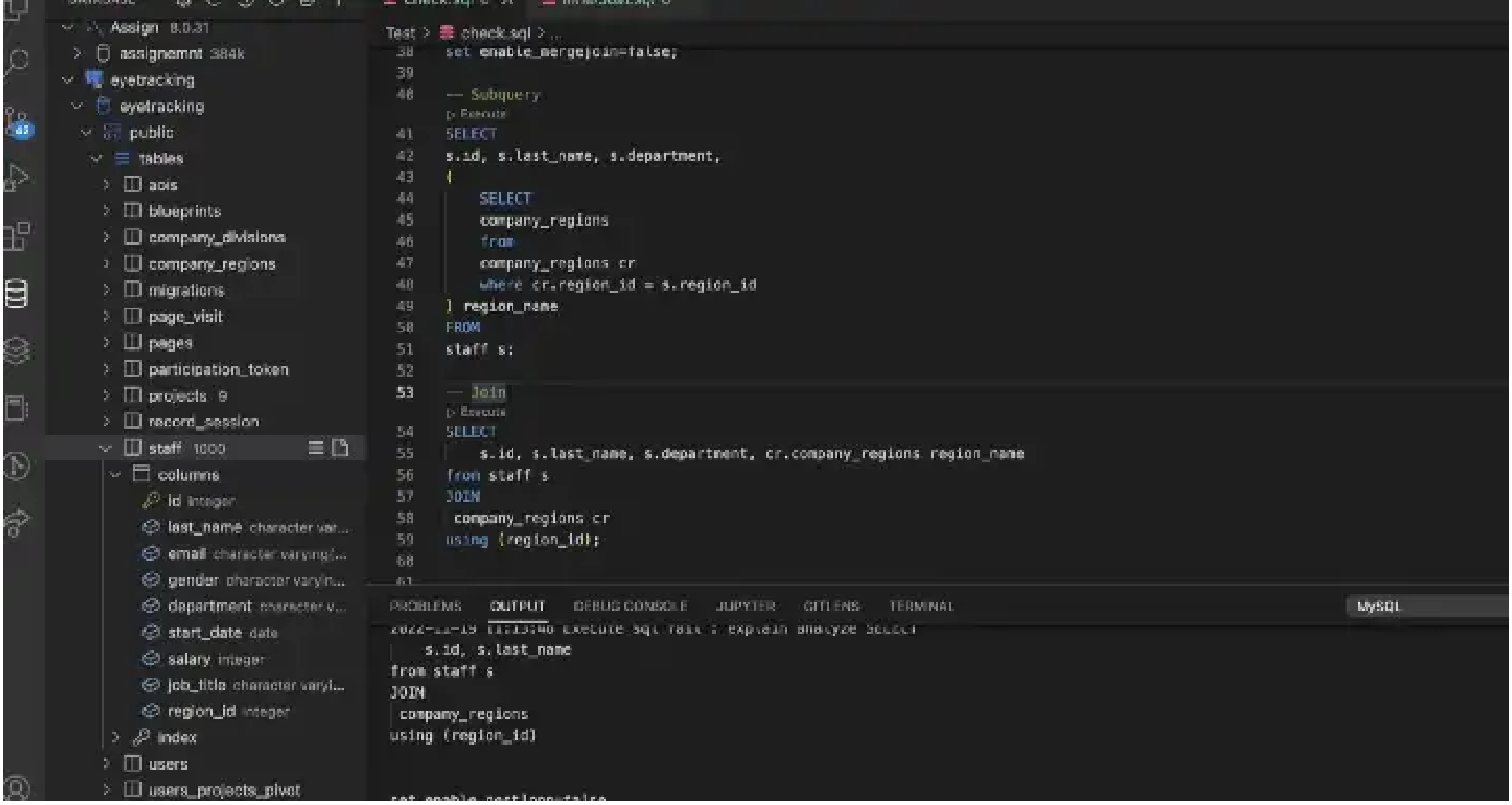
Well if we go by the conventional wisdom, then we should always use joins since they're more efficient.

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