These queries were run on the ff. specs:

Data:

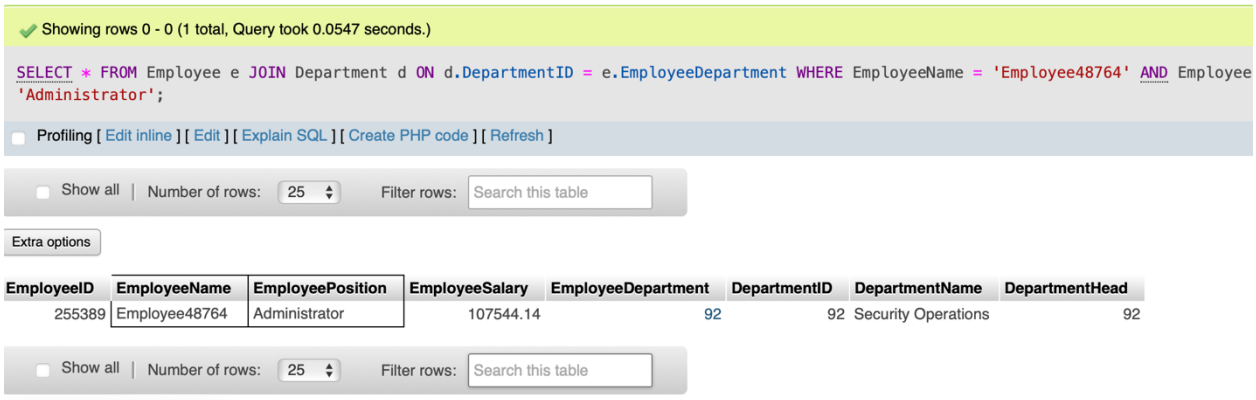
Employee Table: 200,000 rows

Department Table: 100 rows

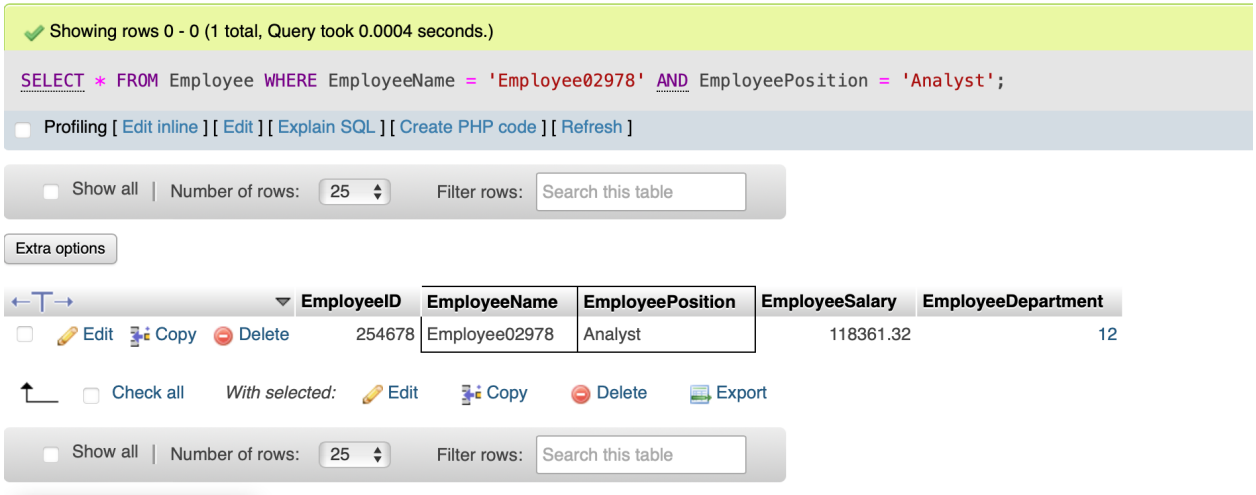
Query # 1

```
SELECT
    *
FROM
    Employee
WHERE
    EmployeeName = 'Employee48764'
    AND EmployeePosition = 'Administrator';
```

Without Index : **0.0547 seconds**



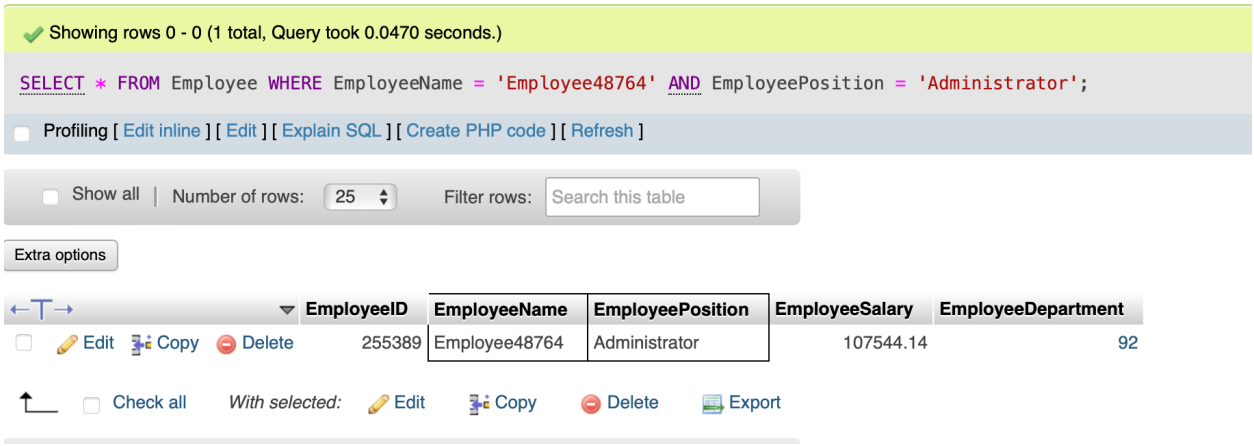
With Index: **0.0004 seconds**



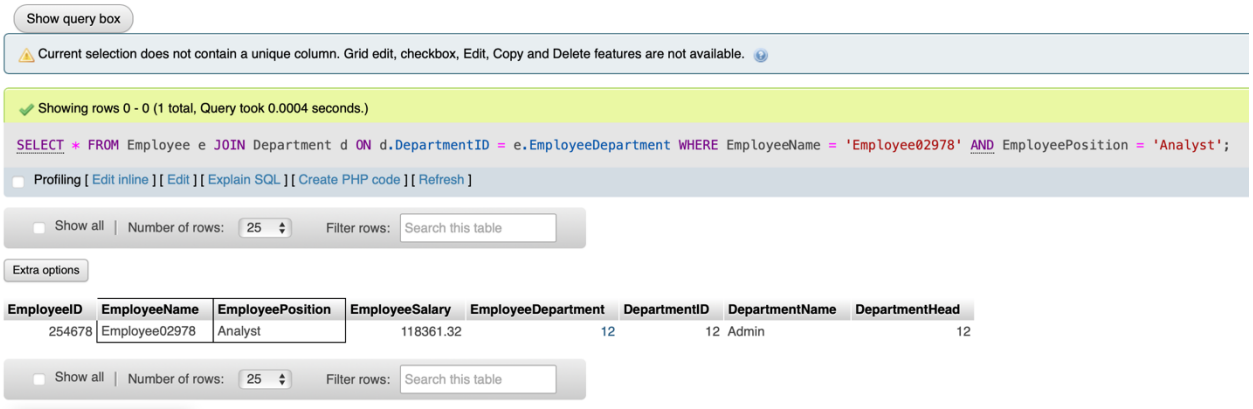
Query # 2:

```
SELECT
    *
FROM
    Employee e
    JOIN Department d ON d.DepartmentID = e.EmployeeDepartment
WHERE
    EmployeeName = 'Employee48764'
    AND EmployeePosition = 'Administrator';
```

Without Index : **0.0470 seconds**



With Index : **0.0004 seconds**



Query # 3:

```
SELECT
    *
FROM
    Employee e
    JOIN Department d ON d.DepartmentID = e.EmployeeDepartment
GROUP BY
    EmployeeName
HAVING
    EmployeeName = 'Employee48764'
    AND EmployeePosition = 'Administrator';
```

Without Index : **1.12 seconds**

With Index **: 1.12 seconds**

We can clearly see how indexes improve the performance of queries. Although in terms of milliseconds, if we convert this time to real-life it already makes a huge difference. In the third query where we used group by, they performed about the same. This tells us that depending on how we structure our queries, we could also get varied results.

a. What is indexing in a database, and why is it important?

- Indexing in a database is where we tell our database engine to create another table based on the original table where we created our index from. This table contains pointers to the actual data and these pointers are sorted in ascending order. This is important especially for tables that contain large datasets because if the data are sorted, it would be easier to scan through elements through a binary search, one of the most efficient searching algorithms. If the data is sorted, we won't have to sequentially go through each row like looking for a word in the dictionary. If we want to find the meaning of "whimsical", we won't go to the first page and start looking from there. I'd probably go to the latter part of the dictionary where the "w" words can be found. This saves me a lot of time and resources.

b. Explain the difference between clustered and non-clustered indexes.

- A clustered index is an index whose order matches the order of the original table. While the non-clustered index is an index whose order does not match the original table. Our db engine automatically creates a clustered index in our table once we set a column or columns as our primary key. This becomes the only clustered index because it's not possible to have 2 clustered indexes. If the primary key index matches the original order in the table, then an index from another column won't be able to match the original order unless in very rare impractical occasions where the data in the column of that table is already sorted. And in that case, at some point this column will become unsorted or else we wouldn't need an index anymore.

c. When should you consider creating an index on a column? Are there any scenarios can have a negative impact?

- We should consider creating an index on a column if we perform queries on these columns especially those columns that we put in our WHERE clause. These indexes impact search efficiently significantly. Although indexing has it's advantages, we also have to be very careful when and when not to use them. Creating indexes mean that we would eat up more memory and this would add additional overhead when we insert, update, or delete data(because when we insert, update, or delete from the original table, we also need to insert, update, or delete from our index) from the table making maintenance a bit more costly. So, we shouldn't create indexes for those columns we know we won't be using when searching or scanning for particular data.

d. How does indexing affect data modification operations, such as insert, update, and delete?

- Indexing adds additional overhead to insert, update, and delete operations because we also need to re-perform these operations in the indexes and not just in the actual table.

e. What are some alternative data structures or indexing techniques used in modern databases?

- There are multiple alternative data structures and indexing techniques in modern databases. Some of these include B-trees using the tree data structure. Another one is hash indexes to map keys to locations where the data is stored. Through this we don't have to loop through all the data since we already know which memory location it's stored in using the hash key. We also have hybrid indexes, inverted indexes, and a lot more.

Understanding Isolation Levels in Databases

If only every database in this world is accessed by just one user/client, then we wouldn't have to pain our heads with isolation levels. However, hundreds, and thousands of users/clients need access to our databases at the same time so how do we handle this? How do we make sure that each of our users/clients are able to interact with the database and get the results that we need? This is where Isolation levels come in. Isolation levels are rules that we set on our transaction (A transaction could be a query or a group of queries to our database which could be initiated by a user/client of the database) telling how the queries inside this transaction should be executed. One typical example would be if one transaction from a user in Los Angeles is trying to read data from our database while another user in New York is trying to insert a data into our database, how will the transaction from Los Angeles perform the read? Should it include or exclude the data simultaneously added by the user from New York?

One major advantage of Isolation levels is that it allows data consistency. Let's first further understand what a transaction is. As mentioned, a transaction could be a query or a group of queries from a user of the database. In performing these transactions, our databases follow the ALL OR NOTHING rule. Meaning if one of the queries in a transaction fail, it will rollback all the queries that have been executed thus far in that specific transaction. So, if our transaction involves Inserting a row in our table and the second query involves updating another row and say the ID of the row being updated is not present, this throws an error so even if the data from the previous query has been inserted, it will be rolled back because not all the queries in that transaction executed successfully. The only way to make sure that changes can't be rolled back is if that transaction has been committed.

Going back to our LA and New York situation, we can set the isolation level of LA's transaction to READ COMMITTED. Meaning, it will not read the data that the user from New York inserted because it hasn't been committed yet and it can still be rolled back. If we didn't set this rule, LA user could have already read that data only to find out later that the actual data does not exist because the New York user rolled back it's transaction. Isolation levels also allow concurrency where multiple clients can access the database at the same time, so the user does not have to wait for other users to finish with their queries. And lastly, since we allow these concurrent accesses, our system performs faster and could quickly respond to user's requests. However, the higher the level we go up in our Isolation levels, the more performance will be impacted. Let's discuss this in detail by comparing the different Isolation levels.

Isolation Level	Dirty Read	Non- Repeatable Read	Phantom Read	Performance (1- 4 w/ 1 being the slowest)
Read Uncommitted	YES	YES	YES	4
Read Committed	NO	YES	YES	3
Repeatable Read	NO	NO	YES	2
Serializable	NO	NO	NO	1

If you noticed, Isolation levels are specifically created for READ queries because these are the only queries that involve fetching data. Also to make things clearer, transactions from multiple users/clients are executed at the same time, meaning the DB engine does not wait to finish all the queries of one transaction before performing the queries of another transaction thus there's a good chance that conflicts may occur.

READ UNCOMMITTED – This Isolation level reads all rows of data even if those rows come from transactions that haven't been committed yet. Since these haven't been committed, chances are they could be rolled back and now we've read data that doesn't even actually exist. This is called a DIRTY READ. This does not align with our goal of data consistency or data integrity, so we usually don't want this. However, this is the Isolation level that performs best because it has no restrictions, so a decision must be made between data consistency vs query performance.

Real-Life Example(Read Uncommitted):

Say for example we are creating an app to track website activity across a thousand users in real-time where every click, every refresh, and every action is recorded. Let's assume one user clicks on a link that redirects to another web page. The click has been logged in our database but that click log hasn't been committed yet because the webpage the user is supposed to go to is still loading. The webpage ends up not loading at all so the transaction is rolled back, and the click log has been deleted. However, in this tracking app, it is critical to get relevant and timely data as the website activities are being executed by the thousands of users. One inconsistent click won't really affect the charts or graphs that will be created out of these activity data. What's more important is that we can generate these charts as soon as possible. In these cases, we will allow dirty reads because performance is more beneficial over a few random data inconsistencies that have little to no effect on the graphs and charts that will be used by the stakeholders of the website tracking app to make timely decisions.

READ COMMITTED – This Isolation Level only reads rows of data that have been committed. Say for example we have a table with rows a, b, and c. By the time a user inserts row d but does not commit the transaction yet, and when there is another transaction that performs a read, it doesn't include that uncommitted row of data. This makes sure that it does not read rows that might be rolled-back at any point in the future which ensures consistency in the data read and the actual table data. This is stricter compared to Read Uncommitted but it also comes with performance costs. When a Read Committed transaction reads data, it acquires a lock on the data so it doesn't allow any query to operate on that data. After that data is read, it releases the lock for other transactions to modify. Managing these locks can introduce performance overheads. Also, when another transaction say an update query is operating on a data, it acquires an exclusive lock to that data and does not release it until the transaction has either been committed or rolled-back. Because our isolation level says read committed, it will take time to check if the data being read is currently being exclusively locked and if it is, it will make sure it reads the latest committed data only which again adds to the performance overhead.

Aside from the performance overhead, there is another pitfall. Assuming that we have two read queries within that one transaction, the first read is performed while the insert hasn't been committed yet so it reads a,b,c. However, the second read was performed and by that

time, the inserted row d has already been committed so this read query returns a,b,c,d. We end up with two different read queries in the same transaction returning different inconsistent data. Despite these pitfalls, Read Committed makes us a 100% sure that the data we are reading exists in the database thus promoting data integrity and consistency and this, in most applications is more important than the performance overhead it introduces.

Real-Life Example (Read Committed)

A common example where Read Committed is crucial is in banking systems/apps. A DB transaction is started when someone withdraws from an ATM and while the withdrawal is being processed, another transaction is being created to read the current balance of the account. If the withdrawal is still processing, this transaction has not been committed yet so the user viewing the account balance doesn't see a deduction from the account. If Read Uncommitted was used here, the balance would already be subtracted and if for some reason the ATM withdrawal fails, the user has already seen the subtracted account balance which will trigger a chain of "complaint to the bank" events which every bank does not want. Read Committed is the most commonly used isolation level because it strikes the balance between data consistency and performance.

REPEATABLE READ – This isolation level addresses the issue in Read Committed where a transaction might have 2 read queries and the first query returns a result that does not include the uncommitted queries of another transaction but by the time we reach the 2nd read query, the queries of another transaction was already committed so it returns a different set of data. What repeatable read does is that even if in the 2nd read query, a committed transaction already took place thus changing the actual state of our data, it will still read and return the same data that it read during the first read query where the query from another transaction was not committed yet. Reusing our example above, if the first query reads a,b,c after an uncommitted insert d query was executed, the 2nd read query will still return a,b,c even if by that time the insert d query from another transaction was already committed. However, repeatable read still has a pitfall. It cannot get rid of phantom reads. We will be discussing more about phantom reads in the Serializable section.

Real-Life Example (Repeatable Read):

In healthcare management systems, patient's records like lab results and vital signs could be updated from time to time depending on the condition of the patient. However, if we need to generate a report to determine the condition of the patient in a specific time out of these data, we can't have our vital signs data changing while the report is being generated because this will alter the accuracy and timeliness of the medical report. In this scenarios, repeatable read is very important because it makes sure it reads the same data it read the first time thus producing accurate reports.

SERIALIZABLE – Serializable solves all the problems of the previous isolation levels. It solves dirty reads, non-repeatable reads, and phantom reads. However, it too is the most costly in terms of performance so this must be chosen with careful consideration, all things considered. Let's take a deeper dive on phantom reads. Phantom reads happen when a query in a transaction returns a set of rows and when a new row is inserted or a row is deleted by another transaction, by the time our first transaction performs another read, the result set is now different. Repeatable read cannot prevent this from happening because Repeatable Read only locks individual rows and it can't lock a set of rows so the new rows inserted or rows deleted will be executed. To solve this, serializable locks a whole range of rows affected by the transaction so that no other transaction could perform operations on it. This make it seem like the transactions are executed sequentially because all other transactions will have to wait until the whole Serializable transaction finishes. This is what makes this isolation level slow.

Real-Life Example (Serializable)

One Serializable example in real-life is the flight seat reservation system. It's important for airline companies to sell seats that are available. So to make sure that a customer books an available seat, the reservation system should only read data once another transaction from another customer finishes booking. The performance cost on this real-life scenario is something that's outweighed by the benefit because booking seats on a plane are not life-death situations and people have much options to choose from.